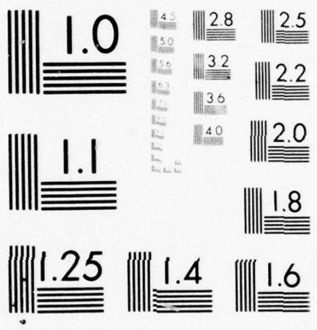

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

2

27332-6921-026

REVS USERS MANUAL

SREP FINAL REPORT - VOLUME II

CDRL C005

1 AUGUST 1977

Prepared For
BALLISTIC MISSILE DEFENSE
ADVANCED TECHNOLOGY CENTER
DASG60-75-C-0022

TRW.
DEFENSE AND SPACE SYSTEMS GROUP
HUNTSVILLE, ALABAMA

TRW

TITLE: REVS USERS MANUAL DATE: 30 SEPTEMBER 1977

DOCUMENT NO: 27332-6921-026

REVISION: A

REASON FOR CHANGE:

This revision documents the ARC CDC 7600 installation of REVS.

INSTRUCTIONS:

To update this manual make the following changes.

AFFECTED PAGES:

iv, x, xi, xiii
1-1
2-13
2-15 (delete)
4-9
5-1
6-25, 6-42
7-5, 7-21
9-1, 9-3 through 9-15
9-16 through 9-18 (add)
10-1 through 10-6 (add)
D-4
D-8
R-1

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CDRL C005 (Volume II)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER (9)
4. TITLE (and Subtitle) (6) REVS Users Manual. (SREP Final Report, Volume II).		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report,
		6. PERFORMING ORG. REPORT NUMBER (14) TRW-27332-6921-026-VOL-2
7. AUTHOR(s) (10) M. E. Dyer, ~~et al~~ L. J. Gunther, R. W. Smith, W. E. Benoit P. N. Bergstresser		8. CONTRACT OR GRANT NUMBER(s) (15) DASG60-75-C-0022
9. PERFORMING ORGANIZATION NAME AND ADDRESS TRW Defense and Space Systems Group 7702 Governors Drive, West Huntsville, Alabama 35805		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 6.33.04.A
11. CONTROLLING OFFICE NAME AND ADDRESS Ballistic Missile Defense Advanced Technology Center, P.O. Box 1500, Huntsville, AL 35807		12. REPORT DATE (11) 1 August 1977
		13. NUMBER OF PAGES 432
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 418p.		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Cleared for public release - distribution unlimited.
Reference BMDSC-CRS letter dated 8 March 1977.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Requirements Engineering and Validation System
Requirements Specification Language
Automated Simulation Generation
Automated Documentation
Language Processors
Software Requirements Engineering

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This manual describes and provides instructions for using the Requirements Statement Language (RSL) and the Requirements Engineering and Validation System (REVS). RSL is a flow oriented language for stating software requirements in a clear, non-ambibiguous form. REVS includes a translator for RSL, a data base for maintaining the description of a system's software requirements, an interactive graphics input and display system, and a set of tools for analyzing the requirements data base for consistency, completeness, and logical integrity. REVS also includes an automated simulation generation capability which (cont'd)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

407 674

1B

20. (Cont'd)

allows simulations to be generated directly from the statement of requirements as written in RSL.

27332-6921-026

REVS USERS MANUAL

SREP FINAL REPORT - VOLUME II

CDRL C005

1 AUGUST 1977

Prepared For

BALLISTIC MISSILE DEFENSE
ADVANCED TECHNOLOGY CENTER

DASG60-75-C-0022

TRW.

DEFENSE AND SPACE SYSTEMS GROUP
Huntsville, Alabama

27332-6921-026

REVS USERS MANUAL

SREP FINAL REPORT - VOLUME II

CDRL C005 1 AUGUST 1977

Principal Authors:

M. E. Dyer
L. J. Gunther
R. W. Smith
W. E. Benoit
P. N. Bergstresser
D. C. Bixler
W. G. Heckler
G. C. Hitt
D. E. McQueen (AIC)

Approved By:

[illegible]
L. R. Marker, Manager
Software Requirements
Engineering Program

M. E. Dyer
M. E. Dyer, Manager
SREP Software and Language
Development

James E. Long
James E. Long, Manager
Huntsville Facility

Prepared For

BALLISTIC MISSILE DEFENSE
ADVANCED TECHNOLOGY CENTER

DASG60-75-C-0022

TRW®
DEFENSE AND SPACE SYSTEMS GROUP
Huntsville, Alabama

RECORD OF REVISIONS

REVISION	DATE	DESCRIPTION
A	8/30/77	Documents the ARC CDC 7600 installation of REVS.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

LIST OF TABLES

1.0 INTRODUCTION

This users manual provides the operating instructions for the Requirements Engineering and Validation System (REVS) software and the definition of the Requirements Statement Language (RSL) which is processed by REVS. This software system and language provide unique capabilities to write, analyze, simulate, and document software requirements. Although they were designed to meet the needs of Ballistic Missile Defense (BMD) systems and other large weapon systems with imbedded real-time software, they are applicable to a broad range of applications. RSL and REVS provide a degree of precision, automation, and confidence in software requirements development unattainable by conventional means.

This manual is organized into five parts to facilitate use by readers at all levels of familiarity with the material beyond a basic understanding of the underlying approach:

- Part I (Section 2) provides an introductory overview of the software and the language.
- Part II (Section 3) describes the concepts of RSL.
- Part III (Sections 4 through 8) describes the RSL syntax and the commands and operating instructions for each of the REVS functions.
- Part IV (Sections 9 and 10) describes the job control language and the installation unique characteristics for REVS.
- Part V (Appendices) provides background and reference material. This material includes an explanation of the extended Backus-Naur Form (BNF) used to define language constructs and various summaries of RSL. The appendices also contain the syntax of the REVS control language and explanations of REVS messages and diagnostics. This material provides the quick reference needed by users of RSL who are already familiar with the underlying concepts and by users of REVS who are familiar with the capabilities of each of the REVS functions.

As with all software users manuals, this manual presents the instructions for using the software and language, but does not contain an explanation of how to apply these capabilities to the development of software requirements. The engineering application of these capabilities is described in the Software Requirements Engineering Methodology, which is Volume I of this report.

2.0 OVERVIEW OF RSL AND REVS

The Requirements Statement Language (RSL) provides the user with the ability to define software requirements in a form which assures unambiguous communication of explicit, testable requirements. RSL combines the readability of English with the rigor of a computer-readable language. The Requirements Engineering and Validation System (REVS) provides facilities for translating, storing, analyzing, simulating, and documenting requirements written in RSL. Through the use of RSL and REVS, the engineer can verify the completeness and consistency of a software specification with a high degree of confidence.

The common practice of organizing software requirements into a hierarchy of functions, subfunctions, etc., while superficially appealing, leads to difficulties in both the expression and verification/validation of the requirements. This is due in part to the fact that such an organization does not fit the basic input-process-output nature of data processing, and in part to the fact that a hierarchical tree of arbitrarily defined "functions" does not have a sufficiently rigorous mathematical basis to allow automated analysis of completeness and consistency properties of the resulting specification. To avoid these difficulties, RSL and REVS are based on the concept of processing flow. Software requirements written in RSL are formulated in terms of a mathematical network (graph model) called a Requirements Network (R-Net). This approach provides several advantages:

- Describing the required processing in terms of a "logic diagram" of the system is natural to most engineers.
- The mathematical properties of an R-Net allow automated analysis for consistency and completeness through the application of graph theory.
- The flow orientation of an R-Net allows automated generation of simulations directly from the stated requirements.

The remainder of this section presents an overview of the concepts of R-Nets, the concepts of RSL, and the capabilities of REVS.

2.1 REQUIREMENTS NETWORKS (R_NETS)

Flows through a system are specified in RSL as Requirements Networks called R_NETs. R_NET flow structures consist of nodes which specify required processing operations and connecting arcs. The basic nodes are INPUT_INTERFACEs, OUTPUT_INTERFACEs, and required processing activities called ALPHAs. Through the use of these basic nodes, the required paths of processing can be specified. For example, if data is to be input to the data processor through an INPUT_INTERFACE called A, processed by a processing step (ALPHA) called B, then processed by an ALPHA called C, and the result output through an OUTPUT_INTERFACE called D, then the required processing path can be specified by listing the sequence of operations:

```
INPUT_INTERFACE:  A
ALPHA:  B
ALPHA:  C
OUTPUT_INTERFACE:  D
```

This simple R_NET is illustrated graphically in Figure 2-1.

In the above example, the sequence B-C means that those processing steps must be performed in the indicated sequence. In many cases, the actual order of processing is immaterial. This is specified through the use of an AND node as shown in Figure 2-2. This structure means that both B and C must be performed after receipt of data through A and before the result is output through D, but B and C are sequentially independent and may be performed in any order (or in parallel).

Most systems also require the specification of decision (control) points. Thus, in the above example, if B is to be performed under some circumstances (depending on the value of the input data for example) and C is to be performed otherwise, a decision point and its attendant decision criterion must be specified. This is specified in an R_NET through the use of an OR node as illustrated in Figure 2-3. The second OR node following B and C means that processing is to continue (i.e., output result through D) if processing on any input branch has been completed.

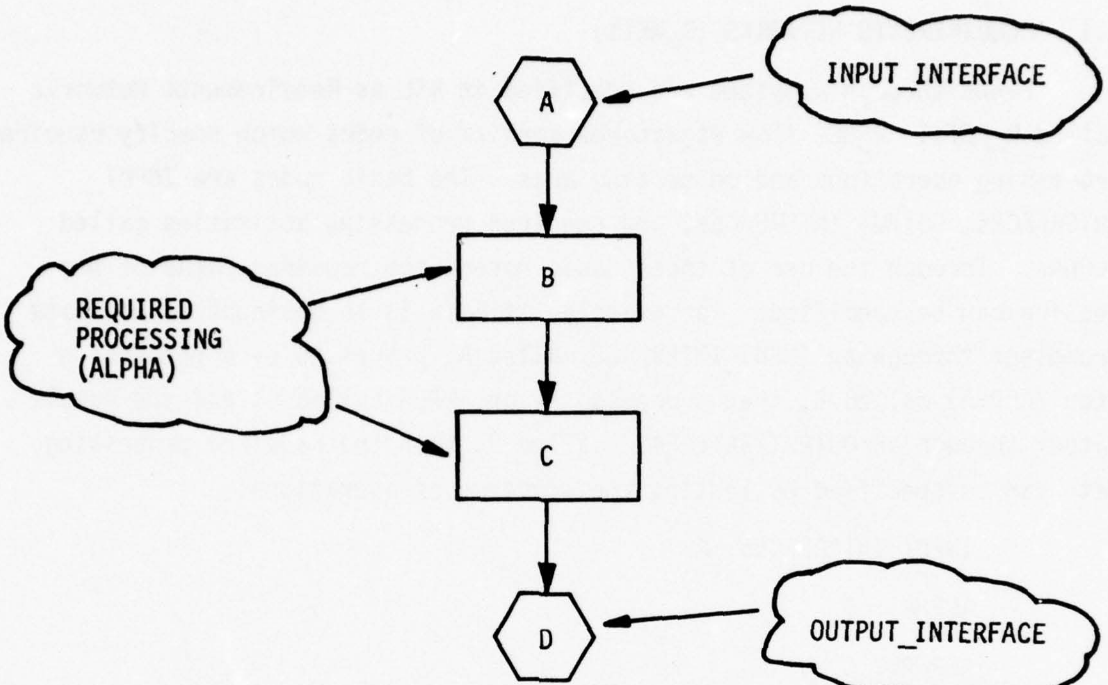


Figure 2-1 An Elementary R_NET

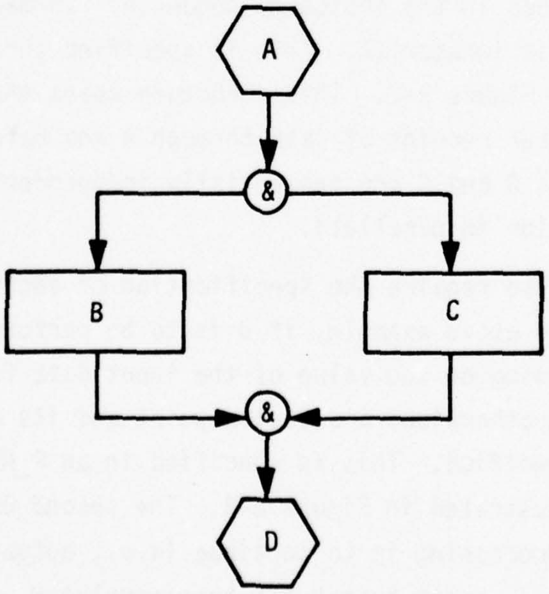


Figure 2-2 Use of AND Nodes in R_NETs

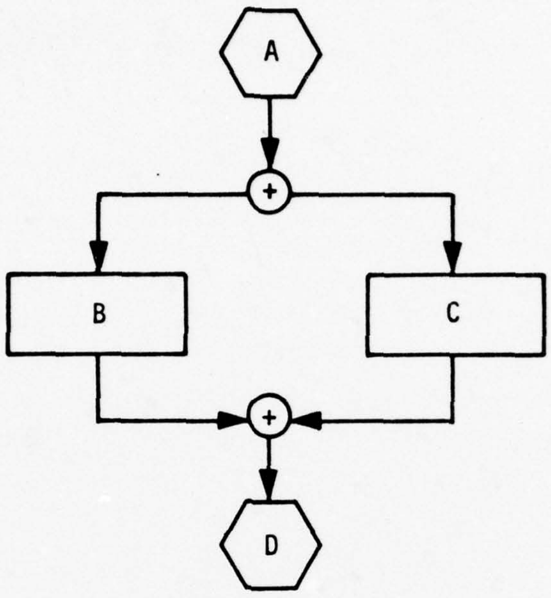


Figure 2-3 Use of OR Nodes in R_NETs

Through the use of the three basic nodes plus AND and OR nodes, complete, complex processing requirements can be readily specified. Other nodes are provided to specify selection of data to be processed (SELECT, FOR EACH), "test points" for specifying performance requirements (VALIDATION_POINTs), internal controls (EVENTs), and detailed processing flows (SUBNETs). These concepts are described in detail in Section 3.

2.2 REQUIREMENTS STATEMENT LANGUAGE (RSL)

RSL is a machine-readable, English-like language for stating software requirements. The basic structure of RSL is very simple and is based on four primitive language concepts: elements, attributes, relationships, and structures.

Elements

Elements in RSL correspond roughly to nouns in English. They are those objects and ideas which the requirements analyst uses as building blocks for his description of the system requirements. Each element has a unique name and belongs to one of a number of classes called element types. Some examples of standard element types in RSL are ALPHA (the class of functional processing steps), DATA (the class of conceptual pieces of data necessary in the system), and R_NET (the class of processing flow specifications).

Attributes

Attributes are modifiers of elements somewhat in the manner of adjectives in English; they formalize important properties of the elements. Each attribute has associated with it a set of values which may be mnemonic names, numbers, or text strings. Each particular element may have only one of these values for any attribute. An example of an attribute is INITIAL_VALUE which is applicable to elements of type DATA. It has values which specify what the initial value for the data item must be in the implemented software and for simulations.

Relationships

The relationship (or relation) in RSL may be compared with an English verb. More properly, it corresponds to the mathematical definition of a binary relation, a statement of an association of some type between two elements. The RSL relationship is non-commutative; it has a subject element and an object element which are distinct. However, there exists a complementary relationship for each specified relationship which is the converse of that specified relationship. ALPHA INPUTS DATA is one of the relationships in RSL; the complementary relationship says that DATA is INPUT to an ALPHA.

Structures

The final RSL primitive is the structure, the RSL representation of the flow graph. Two distinct types of structures have been identified. The first is the R_NET (or SUBNET) structure. It identifies the flow through the functional processing steps (ALPHAs) and is thus used to specify the system response to various stimuli. The second structure type is the VALIDATION_PATH, which is used to specify performance of the system.

Through the use of these four primitive language concepts, a basic requirements language is provided which includes concepts for specifying processing flows, data, processing actions, and timing and accuracy requirements. In addition, informative and descriptive material, and management-related information may be specified. The concepts of this baseline language, consisting of twenty-one element types, twenty-one attributes, twenty-three relationships, and two types of structures, are described in detail in Section 3. Section 5.1 describes the syntax of the language.

RSL can be extended to include additional concepts by defining new element types, attributes, or relationships. This allows the language to be tailored to the needs of a specific problem or project. The explanation of how to extend the language is provided in Section 8.

2.3 REQUIREMENTS ENGINEERING AND VALIDATION SYSTEM (REVS)

The Requirements Engineering and Validation System (REVS) is an integrated system of software which aids in the development, maintenance, validation, and documentation of software requirements. REVS is designed to allow the requirements engineer to state and modify requirements information over a period of time as the requirements are developed. The RSL statements that an engineer inputs to REVS are analyzed, and a representation of the information is put into a centralized data base. This data base is called the Abstract System Semantic Model (ASSM) because it maintains information about the required data processing system (RSL semantics) in an abstract, relational model. Once entered into the ASSM, the requirements are available for subsequent refinement, extraction, and analysis by the REVS software.

From a user point of view there are five major functional capabilities which REVS provides:

- Processing of RSL.
- Interactive generation of Requirements Networks (R_NETs).
- Analysis of requirements and output of requirements in RSL and/or in specially formatted reports.
- Generation and execution of functional and analytic simulators from functional requirements and models or algorithms, and the generation and execution of simulation post processors from analytic performance requirements.
- Processing of extensions to RSL.

REVS and RSL allow the engineer to enter requirements into REVS as they are developed, with REVS accumulating the information in the requirements data base and checking for consistency and completeness as new data is entered. Consequently, although the REVS capabilities may be applied in any order, in general, the user will initially enter RSL and request various analyses to be performed. New entries will be made and analysis repeated until the requirements have been developed sufficiently for a simulation to be meaningful and useful. At that time a simulator and post processor may be generated. The simulator may then be executed numerous times and the data recorded and analyzed. Based on the results, this sequence may be repeated, starting with the modification of requirements already input to REVS or the addition of new ones. The sequence will also be repeated as system requirements change or new requirements are imposed. When the user is satisfied that the requirements are correct, based upon the results of static and dynamic analysis, REVS will document the requirements in a form directly usable in a software requirements specification.

Each of the major capabilities identified above is allocated to a different functional component of REVS. The capabilities and the appropriate functions are described briefly below.

Processing of RSL

The analysis of RSL statements and the establishment of entries in the ASSM corresponding to the meaning of the statements is performed by the RSL translation function of REVS (see Section 5.1). The translation function also processes the modifications and deletions from the data base commanded by RSL statements specifying changes to already-existing entries in the data base. For all types of input processing, the RSL translation function references the ASSM to do simple consistency checks on the input. This prevents disastrous errors such as the introduction of an element with the same name as a previously-existing element or an instance of a relationship which is tied to an illegal type of element. Besides providing a measure of protection for the data base, this type of checking catches, at an early stage, some of the simple types of inconsistencies that are often found in requirements specifications, without restricting the order in which the user adds to or alters the data base.

Interactive Generation of R-Nets

Graphics capabilities to interactively input, modify or display R_NET, SUBNET, and VALIDATION_PATH structures are provided through the REVS Interactive R-Net Generation (RNETGEN) function. RNETGEN permits entry of structures and referenced elements in a manner parallel to the RSL translator and thus provides an alternative to the RSL translator for the specification of the flow portion of the requirements. Using this function, the user may develop (either automatically or under direct user control) a graphical representation of a structure previously entered in RSL. Through the use of the ASSM, the user may work with either the graphical or RSL language representation of a structure; they are completely interchangeable.

The Interactive R-Net Generation facility contains full editing capabilities. The user may input a new structure or he may modify one previously entered. At the conclusion of the editing session, the user may elect to replace the old structure with the modified one. The editing functions provide means to position, connect, and delete nodes, to move them, to disconnect them from other nodes and to enter or change their associated names and commentary. The size of a structure is not limited by the screen; zoom-in, zoom-out, and scroll functions are provided. Details of the RNETGEN capabilities are presented in Section 5.2.

Analysis and Output of Requirements

The Requirements Analysis and Data Extraction (RADX) function provides both static flow analysis capabilities and the capabilities of a generalized extractor system to support both the checking for completeness and consistency in the requirements specification and the development of requirements documentation (see Section 6.0).

The static flow analysis deals with data flow through the R_NETs. The analysis uses the R_NET structure in much the same manner that data flow analyzers for programming languages use the control flow of the program to detect deficiencies in the flow of processing and data manipulation stated in the requirements.

The generalized extractor system allows the user to perform additional analysis and to extract information from the ASSM. The user can subset the elements in the ASSM based on some condition (or combination of conditions) and display the elements of the subset with any appended information he selects.

Information to be retrieved is identified in terms of RSL concepts. For example, if the user wants a report listing all DATA elements which are not INPUT to any ALPHA (processing step), he enters the following commands:

```
SET A = DATA THAT IS NOT INPUT.
LIST A.
```

By combining sets in various ways, he can detect the absence or presence of data, trace references on the structures, and analyze interrelationships established in the ASSM. In analyzing user requests and extracting information from the ASSM, the extractor system uses the definition of the language concepts contained in the ASSM. Thus, as RSL is extended, the extensions and their use in the requirements are available for extraction.

Generation and Execution of Simulators and Post Processors

The automatic Simulation Generation (SIMGEN) function in REVS takes the ASSM representation of the requirements of a data processing system and generates from it discrete event simulators of the system. These simulators are driven by externally generated stimuli; the baseline system generates simulators to be driven by a System Environment and Threat Simulation (SETS) type of driver program which models the threat, the system environment, and the components of a ballistic missile defense system external to the data processing system.

Two distinct types of simulators may be generated by REVS. The first uses functional models of the processing steps and may employ simplifications to simulate the required processing. This type of simulation serves as a means to validate the overall required flow of processing against higher level system requirements.

The other type of simulator uses analytic models; i.e., models that use algorithms similar to those which will appear in the software to perform complex computations. This type of simulation may be used to define a set of algorithms which have the desired accuracy and stability. Real-time feasibility of a system using this algorithm set is not established for any implementation; instead the simulation provides an existence proof of an analytic solution to the problem. Both types of simulations are used to check dynamic system interactions.

The SIMGEN function transforms the ASSM representation of the requirements into simulation code in the programming language PASCAL. The flow structure of each R_NET is used to develop a PASCAL procedure whose control flow implements that of the R_NET structure. Each processing step (ALPHA) on the R_NET becomes a call to a procedure consisting of the model or algorithm for the ALPHA. The models or algorithms are written in PASCAL. The data definitions and structure for the simulation are synthesized from the requirements data elements and their relationships and attributes in the ASSM.

By automatically generating simulators in this manner from the ASSM, the simulations are insured to match and trace to the requirements. New simulators can be generated readily as requirements change; all changes are made to the requirements statements themselves, and are automatically reflected in the next generation of the simulator.

For analytic simulations, SIMGEN also generates simulation post processors based on the statement of performance requirements in the ASSM. Data collected from an analytic simulation can be evaluated using the corresponding post processor to test that the set of algorithms meet the required accuracies.

Both REVS generated simulators and post processors are accessed for execution through REVS functions; the Simulation Execution (SIMXQT) function for simulators, and the Simulation Data Analysis (SIMDA) function for simulation post processors. The REVS generation and execution of simulators and post processors is detailed in Section 7.

Processing Extensions to RSL

An ASSM contains the RSL concepts used to express requirements as well as the requirements. Extensions and modifications to the concepts are processed by the RSL Extension translation (RSLXTND) function of REVS as described in Section 8. The RSLXTND function is actually performed by the same software as RSL translation but is accessed separately to control extensions to the language through a lock mechanism built into the software.

REVS Organization

The above discussion has identified seven functions of REVS: RSL, RNETGEN, RADX, SIMGEN, SIMXQT, SIMDA and RSLXTND. As shown in Figure 2-4, these functions are under the control of a higher level function, the REVS Executive. The Executive presents a unified interface between the user and the different REVS functions. The organization of command input to REVS and the REVS controls available at the Executive level are described in Section 4.

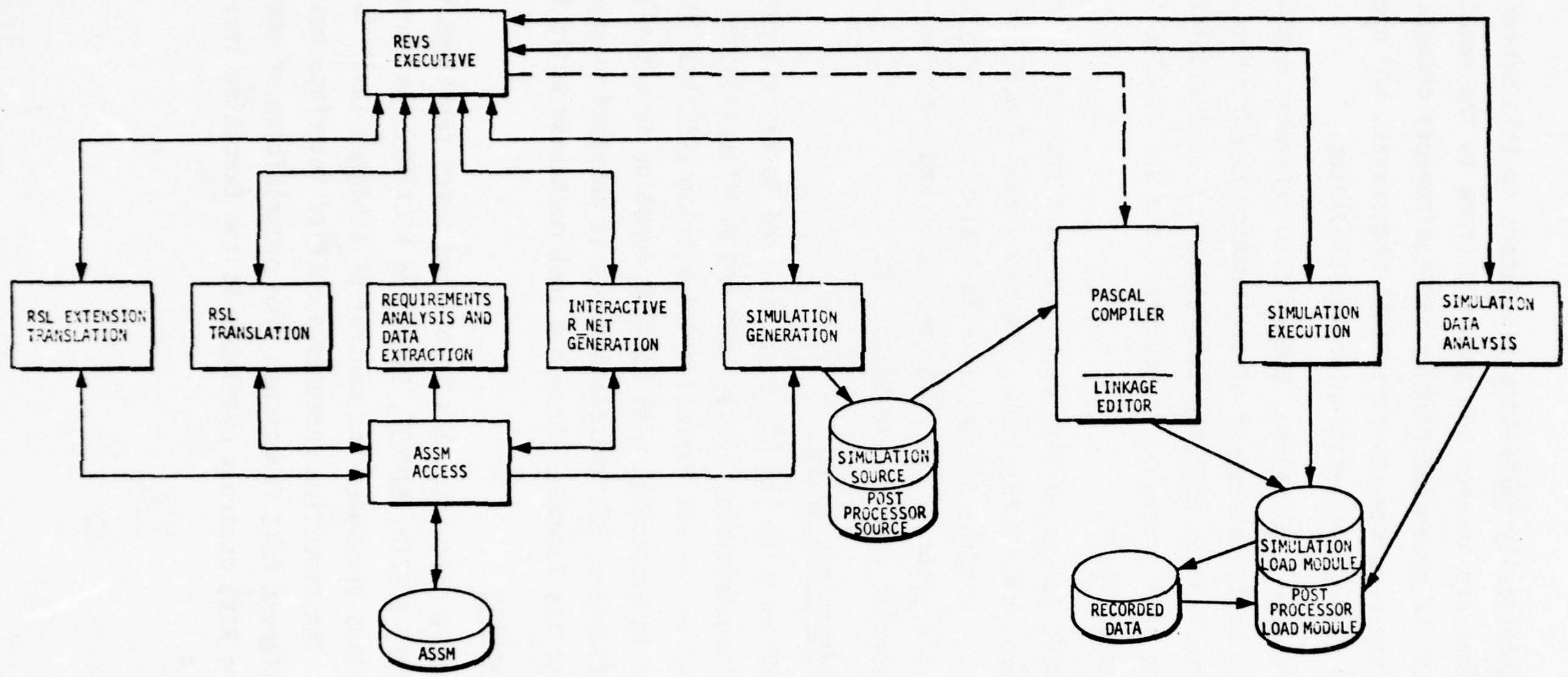


Figure 2-4 REVS Functional Organization

BEST AVAILABLE COPY

installation of REVS operates either from card input (termed off-line mode) or interactively (termed on-line mode) using the Data Disc Color Graphics (ANAGRAPH) Display System Terminal. In the on-line mode, all of the functions described above are available to the user and may be invoked in any order. In the off-line mode, all functions except RNETGEN may be utilized in any order. The NRL installation of REVS operates only in the off-line mode. An explanation of the job control stream required to execute REVS on either of these ASCs is documented in Section 9.[1]

[1]REVS will be operational in July 1977 on a Control Data Corporation (CDC) 7600 at the ARC. This machine also interfaces with the ANAGRAPH terminal. A supplement to this volume documenting REVS operation on the CDC 7600 will be published separately.

3.0 REQUIREMENTS STATEMENT LANGUAGE

RSL is a concept-oriented language based on the four primitives of elements, relationships, attributes, and structures as described in Section 2. The unit for writing requirements in RSL is the element definition. An element definition is very similar to a paragraph in English. It consists of one or more sentences, the first of which is a topic sentence. The topic sentence gives the element type and the name of the element being defined. The other sentences in the definition give the attributes and values, relationships, and structures pertaining to the subject element.

The characteristics of structures and the element types, relationships, and attributes currently defined for writing requirements are presented in this section. The presentation is divided into segments. Loosely, a segment consists of a group of element types, relationships, attributes, and structures which arise from some underlying issue of requirements definition. Element definitions containing concepts from different segments may be intermixed at will. However, the presentation of the predefined concepts of the language will be arranged by segments in order to take advantage of the logical consistency afforded by this view.

This section presents only the concepts of RSL. The syntax for writing RSL element definitions is provided in Section 5.1.

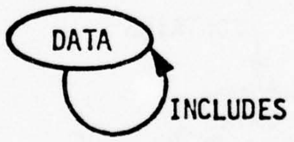
3.1 DATA SEGMENT

The concepts in the Data Segment of RSL address the logical relationships among pieces of information and the interactions of this information with the rest of the software system. Since RSL is used to describe software requirements, not design, the Data Segment concepts address only logical relationships, not physical ones such as access methods, file organizations, etc. The emphasis is on concepts such as hierarchical relationships, use of data by various system components, and access to data using only properties of that data.

3.1.1 Data and Hierarchies

The RSL element type DATA is the class of conceptual pieces of information necessary in the system and includes the meaning of "datum", "data item", "data set", and "data". DATA may have rather obvious relationships with other elements; it may be INPUT to and OUTPUT from ALPHAs (the basic processing steps stated in requirements, see Section 3.2) and RECORDED by VALIDATION_POINTs (the required software test points, see Section 3.4).

DATA may also be organized into hierarchies using the RSL relationship INCLUDES between DATA. This simple data hierarchy is depicted pictorially below.



The requirements engineer might define A to be DATA and also define B to be DATA. He might then define DATA C to INCLUDE both DATA A and DATA B. If he did so, then obtaining C would be exactly equivalent to obtaining both A and B, and we could say that A and B are parts of the hierarchy of C.

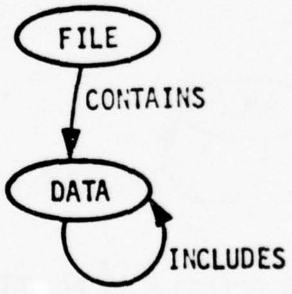
DATA C may also be INCLUDED in DATA D. Thus by the INCLUDES relationship,DATA A, B, and C are part of the hierarchy of D. DATA may be INCLUDED in other DATA in this manner to form any depth of hierarchy as long as no DATA item is repeated (this would specify a loop in the hierarchy, which is not meaningful). Clearly, the only DATA which assume values and are thus required in the real-time software are the DATA comprising the lowest level of a hierarchy (in our example, DATA A and B). The

DATA at higher levels are simply collective names used for clarity and convenience.

Care should be taken in using INCLUDES in conjunction with other relationships. Specifically, a relationship between DATA and a particular element should not be repeated at multiple levels of a hierarchy. To illustrate using our example, if DATA C is INPUT to an ALPHA, the engineer should not also state that DATA A and B are INPUT to the ALPHA. This redundant information is unnecessary -- since inputting C means, by the definition of INCLUDES, that A and B are input. Furthermore, the information may be misleading and confusing to the reader.

3.1.2 Files

In RSL, DATA does not include any repeating lower levels of DATA; C cannot INCLUDE A and n instances of B. Instead, this situation is represented as a DATA item named A and a FILE named D; D CONTAINS instances of DATA B. An RSL FILE, therefore, is a more general concept than a software file since a FILE also subsumes the concepts of arrays and sequences. The relationships CONTAINS and INCLUDES form a hierarchy for defining the constituents of a FILE:



A FILE can CONTAIN any number of DATA; a single DATA, however, is CONTAINED in only one FILE. Each DATA (and its hierarchy) CONTAINED in a FILE is repeated in each instance (record) of the FILE. The number of records in a FILE is not predetermined; instances are entered or removed by the actions of ALPHAs.

A FILE can be INPUT to an ALPHA, implying that the entire FILE is accessible and that one or more instances are retrieved from the FILE; a FILE which is only INPUT may not be altered. The accessing of a FILE can be either:

- ordered (because FILEs are ordered either first-in-first-out -- the default -- or ORDERED least to greatest on any one lowest level DATA CONTAINED in the FILE), or
- associatively (by SELECTing the instances that meet some criterion).

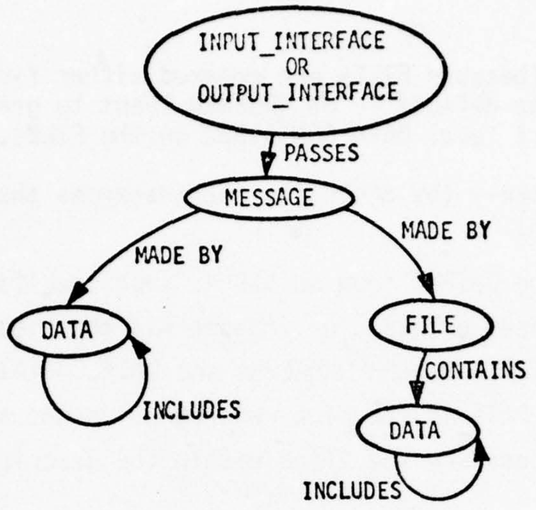
A FILE can also be OUTPUT from an ALPHA; such specification implies that data have been added, deleted, or in some way modified. The relationships INPUTS and OUTPUTS between an ALPHA and DATA CONTAINED in a FILE specify the particular DATA in the FILE records which are affected. The details of the FILE access are specified within the description of the ALPHA.

A FILE can be RECORDED by a VALIDATION_POINT identifying that the entire FILE is to be made available for validation purposes. The relationship RECORDS between a VALIDATION_POINT and DATA CONTAINED in the FILE specifies the particular DATA which are to be RECORDED from each record in the FILE.

3.1.3 Interfaces and Messages

Interfaces exist in real-time software between the data processing subsystem and other components of the system; these interfaces are reflected in two element types in RSL. An INPUT_INTERFACE in RSL denotes an interface through which DATA is communicated into the data processing subsystem. An OUTPUT_INTERFACE is one through which DATA is communicated out of the data processing subsystem. In RSL, each interface CONNECTS the data processing subsystem to some other SUBSYSTEM.

MESSAGEs are the aggregation of DATA and FILEs that are communicated as logical units across interfaces. DATA and FILEs MAKE up MESSAGEs; a single DATA or FILE may MAKE several MESSAGEs. A single INPUT_INTERFACE or OUTPUT_INTERFACE may PASS several different MESSAGEs to or from the data processing subsystem. A given MESSAGE may be PASSED through only one interface. The DATA and FILEs that an INPUT_INTERFACE or OUTPUT_INTERFACE communicates can be ascertained by reference to the definition of all MESSAGEs that PASS through the interface. These relationships, combined with the FILE and DATA hierarchy relationships, form an additional hierarchy for associating information:



An ALPHA on an R_NET path preceding an OUTPUT_INTERFACE FORMS one of the MESSAGEs which the interface PASSES; that is, it indicates which of the legal MESSAGEs the OUTPUT_INTERFACE will PASS when the path is traversed and the interface encountered on an invocation of the R_NET.

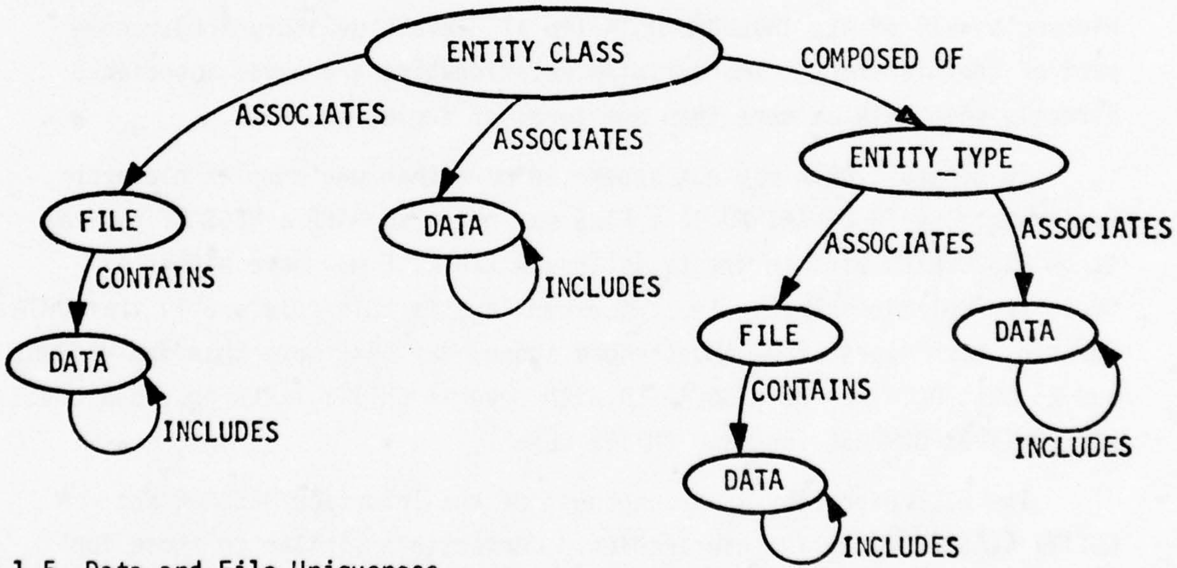
3.1.4 Entity Types and Entity Classes

In real-time control systems the software is generally required to maintain information about objects external to the data processing system. This is represented in RSL using entities. One clear example of an entity in a BMD system is an image -- the thing in space which the BMD system detects, tracks, discriminates, and possibly intercepts. An ALPHA can CREATE and DESTROY the knowledge that an instance of an entity belonging to a particular ENTITY_CLASS exists in the environment.

To reflect the state of the data processing system's knowledge about the entity and thus the "status" of the entity, entities belonging to an ENTITY_CLASS are subsetted into ENTITY_TYPEs; an ENTITY_CLASS is COMPOSED of ENTITY_TYPEs. As the data processing system gathers information about an entity, it may first identify the entity as being of one ENTITY_TYPE, and then another. Instances of a class of entities thus evolve from one type to another, but instances of one class (e.g., images) can never evolve into another class (e.g., interceptors). Each ENTITY_TYPE therefore COMPOSES just one ENTITY_CLASS; an ENTITY_CLASS is COMPOSED of at least one ENTITY_TYPE. An instance belongs to one ENTITY_CLASS after an ALPHA has CREATED it, and it belongs to the last ENTITY_TYPE (within that CLASS) to which an ALPHA SETS it. Eventually, an ALPHA DESTROYS the instance.

ASSOCIATED with each ENTITY_CLASS may be DATA and/or FILEs which are required to be set up whenever the knowledge about a new instance in the ENTITY_CLASS is CREATED by an ALPHA. These DATA and FILEs are maintained for the entity instance through changes to the ENTITY_TYPE until the instance is DESTROYED by an ALPHA.

DATA and FILEs may also be ASSOCIATED with ENTITY_TYPEs. Thus, in addition to the class ASSOCIATED information, an entity instance may have different DATA and FILEs ASSOCIATED with it as it changes ENTITY_TYPE. DATA and FILEs ASSOCIATED with an ENTITY_TYPE may be unique to the type or common to several types composing the same ENTITY_CLASS. As an ALPHA SETS an ENTITY_TYPE for an instance, DATA and FILEs common to the previous and new ENTITY_TYPE retain their values through the change. These relationships, combined with INCLUDES and CONTAINS, form the final data related hierarchy as depicted below:



3.1.5 Data and File Uniqueness

As described above, in addition to simple DATA hierarchies (DATA INCLUDES DATA), RSL provides several complex hierarchies dealing with DATA: FILEs, interfaces and MESSAGEs, and ENTITY_TYPEs and ENTITY_CLASSes. When DATA and FILEs are INPUT to or OUTPUT from an ALPHA, there must be a unique identification of the DATA and FILEs in order for the specification to be unambiguous. Consequently, certain conventions should be followed in writing RSL in order to establish this uniqueness.

The conventions dealing with the individual constructs have been stated in the previous sections introducing the related concepts (e.g., a single DATA item (or its hierarchy) is CONTAINED in only one FILE). In addition there are conventions which apply to all of the constructs and between constructs.

As illustrated in Figure 3-1, a simple data hierarchy may be used as a component of the complex hierarchies. To form complex hierarchies the defining relationships CONTAINS, MADE BY and ASSOCIATES should be used with only two types of DATA:

- simple DATA - DATA which neither INCLUDES DATA nor is INCLUDED in DATA, and
- DATA which forms the top of a hierarchy - DATA which INCLUDES DATA but which is not itself INCLUDED in DATA.

When DATA of the second type listed above is defined as part of a complex hierarchy, all of its INCLUDED DATA (to all levels of inclusion) become part of the hierarchy. The defining relationships are never associated directly with DATA at more than one level of inclusion.

In general, DATA may not appear in more than one complex hierarchy. For example, DATA CONTAINED in a FILE may not also MAKE a MESSAGE nor may it be ASSOCIATED with an entity (although the FILE may have either of these two relationships). The two exceptions to this rule are 1) that DATA (meaning both types of DATA described above) may MAKE more than one MESSAGE and 2) that DATA may be ASSOCIATED with several ENTITY_TYPEs provided the ENTITY_TYPEs COMPOSE the same ENTITY_CLASS.

The FILE hierarchy is a component of the Interface/MESSAGE and ENTITY_CLASS/ENTITY_TYPE hierarchies. Conventions similar to those for DATA apply to FILEs. A FILE cannot appear in both an Interface/MESSAGE hierarchy and an ENTITY_CLASS/ENTITY_TYPE hierarchy. A FILE may MAKE several MESSAGEs. A FILE may be ASSOCIATED with several ENTITY_TYPEs if the types COMPOSE a single ENTITY_CLASS. Finally, DATA and FILEs ASSOCIATED with an ENTITY_CLASS may not also be ASSOCIATED with any ENTITY_TYPEs.

This near absolute separation of the names in data defining hierarchies assures uniqueness of the static DATA definitions. However,

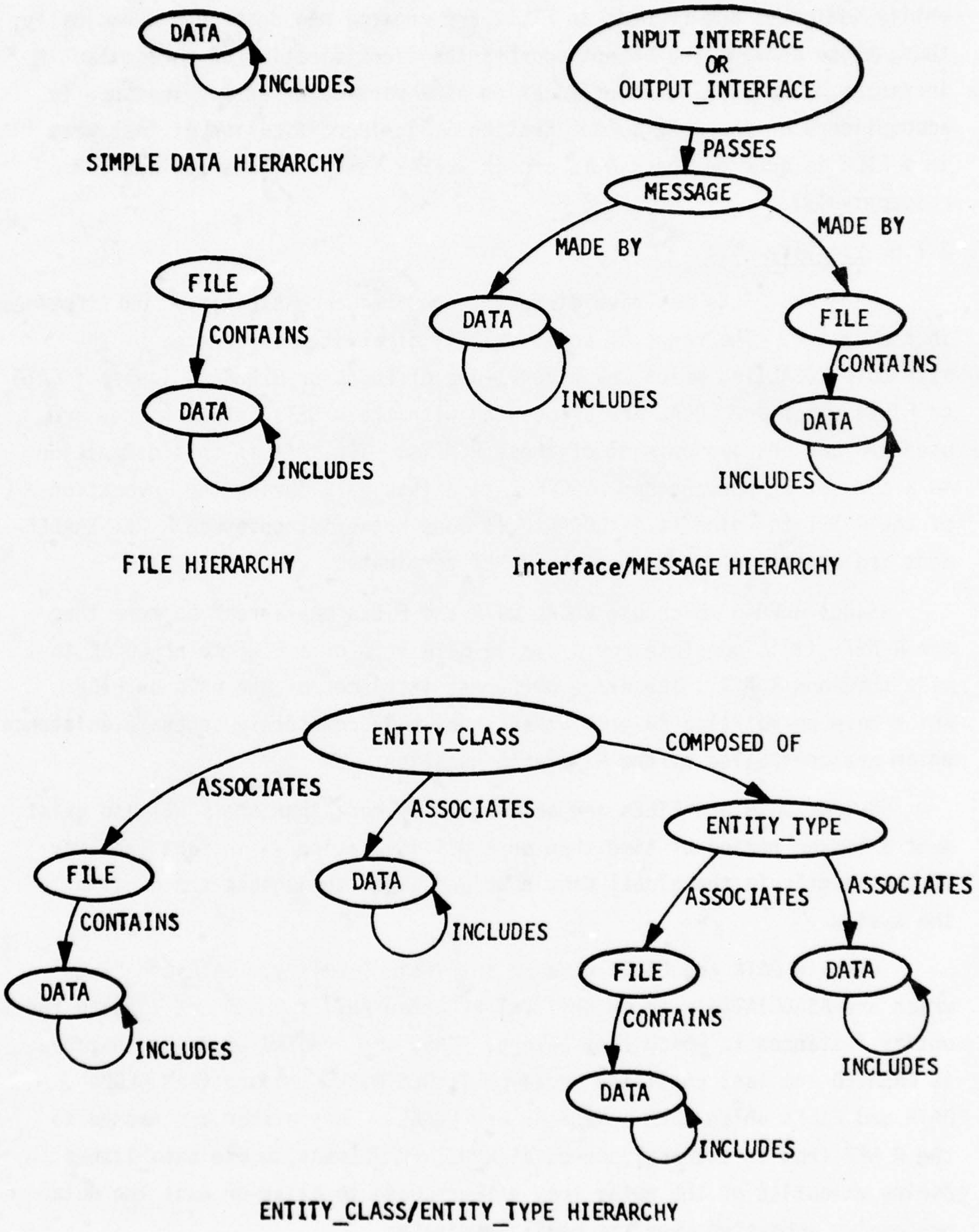


Figure 3-1 Data Relationship Hierarchies

entity instances and records in FILEs are created and destroyed dynamically; thus, these conventions do not address the identification of particular instances of DATA. The identification of a particular entity instance is accomplished on the R_NETs (see Section 3.3); identification of instances in a FILE is done on the R_NETs and in ALPHAs (see Sections 3.3 and 3.2 respectively).

3.1.6 Locality

DATA and FILEs may have different required accessibilities and lifetimes in the system. The range of accessibility of an item is denoted by the attribute LOCALITY, which may have values of LOCAL or GLOBAL. Items of DATA or FILEs which are LOCAL are associated with the R_NETs in which they are used and are unknown outside of these R_NETs. Implicit in this definition is a concept of permanence: LOCAL DATA exists only during the invocation of the R_NET to which it is LOCAL. It does not exist prior to R_NET ENABLE-ment and ceases to exist when the R_NET terminates.

Since ALPHAs which use LOCAL DATA and FILEs may appear on more than one R_NET, it is possible for a single DATA item or a FILE to be LOCAL to more than one R_NET. These are different instances of the DATA or FILE which have no relation to each other; they have completely separate existences which are controlled by the R_NETs in question.

GLOBAL DATA and FILEs are accessible by more than one R_NET and exist over a longer period of time than an R_NET invocation -- in fact they may be permanently in the global data base and exist throughout the duration of the system.

Certain DATA and FILEs have an intrinsic locality. DATA and FILEs which are ASSOCIATED with an ENTITY_TYPE or an ENTITY_CLASS are tied to the entity instances to which they belong. They are created when the instance is CREATED and last until the instance is DESTROYED and are thus GLOBAL. DATA and FILEs which MAKE a MESSAGE are LOCAL. They either are passed to the R_NET from an external source at R_NET ENABLEment, or are established during execution of the nets; they either cease to exist or exit the data processing subsystem when the R_NET terminates.

The lifetime alternatives are therefore:

$$\begin{tikzcd}
 & & \text{MAKES a MESSAGE} \\
\text{DATA} \arrow[r, dash] \arrow[rddd, dash] & \text{LOCAL} \arrow[ur, dash] \arrow[dr, dash] & \\
 & & \text{Transient (during invocation of an R\_NET)} \\
 & & \text{ASSOCIATED with an ENTITY\_TYPE or ENTITY\_CLASS} \\
\text{FILE} \arrow[r, dash] \arrow[ruuu, dash] & \text{GLOBAL} \arrow[ur, dash] \arrow[dr, dash] & \\
 & & \text{Permanent (default)}
\end{tikzcd}$$

Thus, it is necessary to assign the LOCALITY attribute only to those LOCAL DATA and FILEs which do not MAKE a MESSAGE. When assigning LOCALITY to a FILE it is not necessary to also assign LOCALITY to the lowest level of DATA CONTAINED in the FILE. However, if the DATA has LOCALITY it must match that of the FILE.

3.1.7 Typing and Usage

As described in Section 3.3, DATA are used in branching criteria on R_NETs to determine the flow of processing. To understand these branching criteria, TYPE information must be provided about the DATA. The attribute TYPE may have values *REAL*, *INTEGER*, *BOOLEAN*, *or ENUMERATION*. A DATA item with TYPE ENUMERATION has values which are denoted by identifiers given in the RANGE attribute, which is legal only for DATA items which are enumerated. An example of the use of the enumerated type follows:

```
DATA:  COLOR.
TYPE:  ENUMERATION.
RANGE: "RED, BLUE, YELLOW, GREEN".
```

Clearly, for stating requirements the attribute TYPE is assigned to only simple DATA or DATA at the lowest level of a hierarchy (e.g., simple data, FILE, Interface/MESSAGE, or ENTITY_CLASS/ENTITY_TYPE hierarchy).

TYPE information must also be provided for simulation purposes. For simulations, models of the required processing steps are written. These models may be either functional models (termed BETAs) or analytic models (termed GAMMAs). An analytic simulation (one which uses GAMMAs) by definition simulates the lowest level of DATA -- the requirements level. Thus, the TYPE attribute is always assigned for gamma simulations as it is for requirements -- on either simple DATA or DATA at the lowest level of a hierarchy.

A functional simulation (using BETAs) may employ DATA only part way down a hierarchy. That is, the simulation may use one DATA to represent all or a part of a hierarchy. For example, a beta simulation may use the DATA POSITION to represent the DATA X, Y, and Z INCLUDED in POSITION. Thus, in order for the simulation to execute, the TYPE attribute must be assigned to DATA POSITION.

Qualification of the use of a DATA item in simulation and interpretation of TYPE is given by the attribute USE. The value of this attribute may be BETA, GAMMA, or BOTH denoting that the DATA item is the lowest level in the data hierarchy which will be used in the corresponding simulation. The lowest level of DATA must always have either USE GAMMA or USE BOTH. Thus in our example, DATA POSITION would have USE BETA; DATA X, Y, and Z would have USE GAMMA; and DATA POSITION, X, Y, and Z would have specified TYPEs.

3.1.8 Values

In stating software requirements the engineer may specify the attributes UNITS, MAXIMUM_VALUE, MINIMUM_VALUE, INITIAL_VALUE, and RESOLUTION at the lowest level in a hierarchy of DATA. The attribute UNITS is given separately to guarantee explicitness and consistency of units between the other attributes. RESOLUTION specifies the required maximum value of the least significant bit for the DATA in units described in the UNITS attribute.

For BETA level simulations, the attributes UNITS, MAXIMUM_VALUE, MINIMUM_VALUE and INITIAL_VALUE may be defined above the lowest level in the hierarchy for those DATA with USE BETA. These specifications, however, are not meaningful as software requirements.

Clearly the assigned values of the attributes INITIAL_VALUE, MINIMUM_VALUE, and MAXIMUM_VALUE should be consistent with the TYPE of the DATA. They should have appropriate numeric values if the TYPE is REAL or INTEGER. For DATA of TYPE ENUMERATION or BOOLEAN, only INITIAL_VALUE is meaningful. For enumerated DATA, INITIAL_VALUE should be assigned a value identified as legal in the RANGE attribute. For DATA of TYPE BOOLEAN, it should have the value TRUE or FALSE.

3.1.9 Summary of Data Segment Concepts

The element types, relationships, and attributes defined below constitute the Data Segment:

ELEMENT TYPES

```
ELEMENT_TYPE:  DATA
               (* A SINGLE PIECE OF INFORMATION OR SET OF
                  INFORMATION THAT IS EITHER REQUIRED IN THE
                  IMPLEMENTED SOFTWARE OR IS NEEDED FOR
                  DESCRIPTIVE PURPOSES. *).

ELEMENT_TYPE:  ENTITY_CLASS
               (* A GENERAL CATEGORY OF OBJECTS OUTSIDE THE DATA
                  PROCESSING SUBSYSTEM.  THE OBJECTS MAY BE REAL
                  OR PERCEIVED AND ARE THOSE IN THE ENVIRONMENT
                  ABOUT WHICH THE DATA PROCESSING SUBSYSTEM MUST
                  MAINTAIN INFORMATION.  FOR EXAMPLE, AN
                  ENTITY_CLASS MIGHT BE TARGET OR INTERCEPTOR.
                  WHEN THE EXISTENCE OF AN OBJECT IN AN
                  ENTITY_CLASS IS DETERMINED, FILES AND DATA MAY
                  BE TEMPORARILY CREATED TO MAINTAIN INFORMATION
                  ABOUT IT. *).

ELEMENT_TYPE:  ENTITY_TYPE
               (* A SUBSET WITHIN A GENERAL CLASS (ENTITY_CLASS)
                  OF OBJECTS OUTSIDE THE DATA PROCESSING
                  SUBSYSTEM ABOUT WHICH THE DATA PROCESSOR MUST
                  MAINTAIN INFORMATION.  FOR EXAMPLE,
                  ENTITY_TYPES WITHIN THE ENTITY_CLASS TARGET
                  MIGHT BE DETECTION, POTENTIALLY, NON-
                  THREATENING, THREATENING, ETC.  WHEN A
                  PARTICULAR OBJECT IN AN ENTITY_CLASS IS
                  DETERMINED TO BE OF A SPECIFIC TYPE, THE
                  OBJECT CAN BE SET TO THE TYPE AND DATA AND
                  FILES PERTINENT TO OBJECTS OF THAT TYPE
                  TEMPORARILY CREATED TO MAINTAIN INFORMATION
                  ABOUT THE OBJECT. *).

ELEMENT_TYPE:  FILE
               (* AN AGGREGATION OF INSTANCES OF DATA, EACH
                  INSTANCE OF WHICH IS TREATED IN THE SAME
                  MANNER. *).

ELEMENT_TYPE:  INPUT_INTERFACE
               (* A PORT BETWEEN THE DATA PROCESSING SUBSYSTEM
                  AND ANOTHER SUBSYSTEM (E.G., A RADAR) THROUGH
                  WHICH DATA IS PASSED TO THE DATA PROCESSING
                  SUBSYSTEM.  AN INPUT_INTERFACE APPEARS AS THE
                  FIRST NODE OF ONE AND ONLY ONE R_NET
                  STRUCTURE. *).
   STRUCTURE APPLICABILITY:  NET.

ELEMENT_TYPE:  MESSAGE
               (* AN AGGREGATION OF DATA AND FILES THAT PASS
                  THROUGH AN INTERFACE AS A LOGICAL UNIT. *).
```

```
ELEMENT_TYPE:  OUTPUT_INTERFACE
               (* A PORT BETWEEN THE DATA PROCESSING SUBSYSTEM
                  AND ANOTHER PART OF THE SYSTEM (E.G., A RADAR),
                  THROUGH WHICH DATA IS PASSED TO THE OTHER
                  SUBSYSTEM.  AN OUTPUT_INTERFACE MAY APPEAR ON
                  AN R_NET OR SUBNET STRUCTURE AS THE LAST NODE
                  OF A PATH. *).
   STRUCTURE APPLICABILITY:  NET.

ELEMENT_TYPE:  SUBSYSTEM
               (* A PART OF THE SYSTEM (E.G., A RADAR) WHICH
                  COMMUNICATES WITH THE DATA PROCESSING
                  SUBSYSTEM. *).
```

RELATIONSHIPS

```
RELATIONSHIP:  ASSOCIATES
               (* IDENTIFIES WHICH DATA AND FILES COME INTO
                  EXISTENCE WHEN A DATA PROCESSING STEP (AN
                  ALPHA) EITHER CREATES AN INSTANCE OF AN
                  ENTITY_CLASS OR SETS THE ENTITY_TYPE OF AN
                  INSTANCE OF AN ENTITY_CLASS.  DATA AND FILES
                  CAN BE ASSOCIATED WITH ONLY ONE ENTITY_CLASS.
                  DATA AND FILES MAY BE ASSOCIATED WITH SEVERAL
                  ENTITY_TYPES PROVIDED THE ENTITY_TYPES
                  COMPOSE THE SAME ENTITY_CLASS.  DATA AND FILES
                  THAT ARE ASSOCIATED WITH AN ENTITY_TYPE OR
                  ENTITY_CLASS MAY NOT ALSO MAKE A MESSAGE.  DATA
                  THAT IS ASSOCIATED WITH AN ENTITY_TYPE OR
                  ENTITY_CLASS MAY NOT ALSO BE CONTAINED IN A
                  FILE. *).
   COMPLEMENTARY RELATIONSHIP:  ASSOCIATED ("WITH").
   SUBJECT ELEMENT_TYPE:  ENTITY_CLASS
                          ENTITY_TYPE.
   OBJECT ELEMENT TYPE:  DATA
                         FILE.

RELATIONSHIP:  COMPOSES
               (* IDENTIFIES TO WHICH ENTITY_CLASS AN ENTITY_TYPE
                  BELONGS.  AN ENTITY_TYPE COMPOSES ONLY ONE
                  ENTITY_CLASS; AN ENTITY_CLASS IS COMPOSED OF AT
                  LEAST ONE ENTITY_TYPE. *).
   COMPLEMENTARY RELATIONSHIP:  COMPOSED ("OF").
   SUBJECT ELEMENT_TYPE:  ENTITY_TYPE.
   OBJECT ELEMENT_TYPE:  ENTITY_CLASS.
```

```
RELATIONSHIP:   CONNECTS ("TO")
                (* IDENTIFIES WITH WHICH SUBSYSTEM THE
                   INPUT_INTERFACE OR OUTPUT_INTERFACE
                   COMMUNICATES.  AN INTERFACE CONNECTS TO ONLY
                   ONE SUBSYSTEM. *).
   COMPLEMENTARY RELATIONSHIP:  CONNECTED ("TO").
   SUBJECT ELEMENT_TYPE:  INPUT_INTERFACE
                          OUTPUT_INTERFACE.
   OBJECT ELEMENT_TYPE:  SUBSYSTEM.

RELATIONSHIP:   CONTAINS
                (* IDENTIFIES THE MEMBERS OF EACH INSTANCE IN A
                   FILE.  DATA MAY BE CONTAINED IN ONLY ONE FILE.
                   DATA THAT IS CONTAINED IN A FILE MAY NOT ALSO
                   MAKE A MESSAGE NOR MAY IT BE ASSOCIATED WITH AN
                   ENTITY_CLASS OR ENTITY_TYPE. *).
   COMPLEMENTARY RELATIONSHIP:  CONTAINED ("IN").
   SUBJECT ELEMENT_TYPE:  FILE.
   OBJECT ELEMENT_TYPE:  DATA.

RELATIONSHIP:   CREATES
                (* INDICATES THAT THE ALPHA CREATES AN INSTANCE OF
                   THE ENTITY_CLASS(ES).  CREATION OF AN ENTITY
                   INSTANCE IN A CLASS OCCURS IMMEDIATELY AT THE
                   BEGINNING OF AN ALPHA WHICH CREATES THE
                   ENTITY_CLASS.  ONLY ONE NEW ENTITY INSTANCE IS
                   CREATED. *).
   COMPLEMENTARY RELATIONSHIP:  CREATED ("BY").
   SUBJECT ELEMENT_TYPE:  ALPHA.
   OBJECT ELEMENT_TYPE:  ENTITY_CLASS.

RELATIONSHIP:   DESTROYS
                (* INDICATES THAT THE ALPHA DESTROYS AN INSTANCE
                   (THE CURRENTLY SELECTED ONE) OF THE
                   ENTITY_CLASS(ES).  IDENTIFICATION OF THE
                   INSTANCE IS PERFORMED BY A SELECT OR FOR EACH
                   NODE ON A NETWORK.  DESTRUCTION OF THE INSTANCE
                   OCCURS IMMEDIATELY BEFORE COMPLETION OF
                   PROCESSING IN THE ALPHA. *).
   COMPLEMENTARY RELATIONSHIP:  DESTROYED ("BY").
   SUBJECT ELEMENT_TYPE:  ALPHA.
   OBJECT ELEMENT_TYPE:  ENTITY_CLASS.

RELATIONSHIP:   FORMS
                (* INDICATES THAT THE ALPHA ESTABLISHES THE
                   MESSAGE AS THE ONE TO BE PASSED BY THE
                   CORRESPONDING OUTPUT_INTERFACE (THE
                   OUTPUT_INTERFACE WHICH PASSES THE MESSAGE)
                   WHEN THAT INTERFACE IS ENCOUNTERED ON THE NET.
                   AN ALPHA MAY FORM SEVERAL MESSAGES PROVIDED
                   THEY ARE PASSED BY DIFFERENT
                   OUTPUT_INTERFACES. *).
   COMPLEMENTARY RELATIONSHIP:  FORMED ("BY").
   SUBJECT ELEMENT_TYPE:  ALPHA.
   OBJECT ELEMENT_TYPE:  MESSAGE.
```

```
RELATIONSHIP:  INCLUDES
               (* INDICATES A HIERARCHICAL RELATIONSHIP BETWEEN
                  DATA.  IF A INCLUDES B, THEN OBTAINING A WILL
                  OBTAIN B. *).
   COMPLEMENTARY RELATIONSHIP:  INCLUDED ("IN").
   SUBJECT ELEMENT_TYPE:  DATA.
   OBJECT ELEMENT_TYPE:  DATA.

RELATIONSHIP:  INPUTS
               (* IDENTIFIES THE DATA AND FILES USED BY THE
                  ALPHA. *).
   COMPLEMENTARY RELATIONSHIP:  INPUT ("TO").
   SUBJECT ELEMENT_TYPE:  ALPHA.
   OBJECT ELEMENT_TYPE:  DATA
                         FILE.

RELATIONSHIP:  MAKES
               (* INDICATES THAT THE DATA OR FILE IS A LOGICAL
                  COMPONENT OF THE MESSAGE.  A DATA OR FILE MAY
                  MAKE SEVERAL MESSAGES.  DATA AND FILES THAT
                  MAKE A MESSAGE MAY NOT ALSO BE ASSOCIATED WITH
                  AN ENTITY_TYPE OR ENTITY_CLASS.  DATA THAT
                  MAKES A MESSAGE MAY NOT ALSO BE CONTAINED IN A
                  FILE. *).
   COMPLEMENTARY RELATIONSHIP:  MADE ("BY").
   SUBJECT ELEMENT_TYPE:  DATA
                          FILE.
   OBJECT ELEMENT_TYPE:  MESSAGE.

RELATIONSHIP:  ORDERS
               (* INDICATES THAT THE VALUE OF THE DATA IS USED TO
                  ORDER THE INSTANCES OF THE FILE.  A FILE MAY BE
                  ORDERED BY ONLY ONE DATA; THE DATA MAY NOT
                  INCLUDE OTHER DATA AND SHOULD BE CONTAINED IN
                  THE FILE. *).
   COMPLEMENTARY RELATIONSHIP:  ORDERED ("BY").
   SUBJECT ELEMENT_TYPE:  DATA.
   OBJECT ELEMENT_TYPE:  FILE.

RELATIONSHIP:  OUTPUTS
               (* IDENTIFIES THE DATA AND FILES WHOSE VALUES OR
                  CONTENTS ARE MODIFIED BY THE ALPHA. *).
   COMPLEMENTARY RELATIONSHIP:  OUTPUT ("FROM").
   SUBJECT ELEMENT_TYPE:  ALPHA.
   OBJECT ELEMENT_TYPE:  DATA
                         FILE.

RELATIONSHIP:  PASSES
               (* IDENTIFIES THE LOGICAL UNITS OF INFORMATION
                  WHICH ARE PASSED THROUGH THE INTERFACE.  AN
                  INTERFACE MAY PASS SEVERAL MESSAGES; A GIVEN
                  MESSAGE MAY BE PASSED THROUGH ONLY ONE
                  INTERFACE. *).
   COMPLEMENTARY RELATIONSHIP:  PASSED ("THROUGH").
   SUBJECT ELEMENT_TYPE:  INPUT_INTERFACE
                          OUTPUT_INTERFACE.
   OBJECT ELEMENT_TYPE:  MESSAGE.
```

```
RELATIONSHIP:  RECORDS
               (* IDENTIFIES THE PARTICULAR DATA AND FILES WHICH
                  ARE TO BE MADE AVAILABLE AT THE
                  VALIDATION_POINT FOR PERFORMANCE
                  EVALUATION. *).
   COMPLEMENTARY RELATIONSHIP:  RECORDED ("BY").
   SUBJECT ELEMENT_TYPE:  VALIDATION_POINT.
   OBJECT ELEMENT_TYPE:  DATA
                         FILE.

RELATIONSHIP:  SETS
               (* INDICATES THAT THE ALPHA ESTABLISHES AN
                  INSTANCE (THE CURRENTLY SELECTED ONE) OF AN
                  ENTITY_CLASS TO BE OF THE ENTITY_TYPE.
                  IDENTIFICATION OF THE INSTANCE IS PERFORMED BY
                  A SELECT OR FOR EACH NODE ON A NETWORK.  AN
                  ALPHA MAY SET SEVERAL ENTITY_TYPES PROVIDED THE
                  ENTITY_TYPES DO NOT COMPOSE THE SAME
                  ENTITY_CLASS.  THE SETTING OF AN ENTITY_TYPE
                  OCCURS IMMEDIATELY IN AN ALPHA SUBSEQUENT TO
                  ANY ENTITY CREATIONS. *).
   COMPLEMENTARY RELATIONSHIP:  SET ("BY").
   SUBJECT ELEMENT_TYPE:  ALPHA.
   OBJECT ELEMENT_TYPE:  ENTITY_TYPE.
```

ATTRIBUTES

```
ATTRIBUTE:  INITIAL_VALUE
            (* THE INITIAL VALUE A DATA ITEM IS REQUIRED TO HAVE
               IN THE IMPLEMENTED SOFTWARE.  THIS VALUE WILL BE
               ASSUMED BY THE DATA ITEM WHEN IT COMES INTO
               EXISTENCE IN A SIMULATION. *).
   APPLICABLE ELEMENT_TYPE:  DATA.
   VALUE: NAMED.
   VALUE: NUMERIC.

ATTRIBUTE:  LOCALITY
            (* THE ACCESSIBILITY AND LIFETIME OF A DATA OR
               FILE. *).
   APPLICABLE ELEMENT_TYPE:  DATA
                             FILE.
   VALUE: GLOBAL
          (* GLOBAL DATA AND FILES ARE ACCESSIBLE BY MORE THAN
             ONE R_NET AND MAY EXIST THROUGHOUT EXECUTION OF THE
             SYSTEM.  DATA AND FILES WHICH ARE ASSOCIATED WITH AN
             ENTITY_TYPE OR AN ENTITY_CLASS ARE BY DEFINITION
             GLOBAL. *).
   VALUE: LOCAL
          (* LOCAL DATA AND FILES ARE ASSOCIATED WITH THE R_NETS
             IN WHICH THEY ARE USED AND EXIST ONLY DURING THE
             INVOCATION OF THE R_NET TO WHICH THEY ARE LOCAL.
             DATA AND FILES WHICH MAKE A MESSAGE ARE BY
             DEFINITION LOCAL. *).
```

```
ATTRIBUTE:  MAXIMUM_VALUE
            (* THE MAXIMUM VALUE A DATA ITEM MAY ASSUME.  THE
               VALUE IS IN THE UNITS STATED IN THE UNITS
               ATTRIBUTE AND SHOULD BE CONSISTENT WITH THE TYPE OF
               THE DATA. *).
   APPLICABLE ELEMENT_TYPE:  DATA.
   VALUE: NUMERIC.

ATTRIBUTE:  MINIMUM_VALUE
            (* THE MINIMUM VALUE A DATA ITEM MAY ASSUME.  THE
               VALUE IS IN THE UNITS STATED IN THE UNITS
               ATTRIBUTE AND SHOULD BE CONSISTENT WITH THE TYPE OF
               THE DATA. *).
   APPLICABLE ELEMENT_TYPE:  DATA.
   VALUE: NUMERIC.

ATTRIBUTE:  RANGE
            (* THE NAMED VALUES THAT CAN BE ASSUMED BY A DATA WITH
               TYPE ENUMERATION. *).
   APPLICABLE ELEMENT_TYPE:  DATA.
   VALUE: TEXT
   (* THE ALLOWED VALUES ARE SEPARATED BY COMMAS. *).

ATTRIBUTE:  RESOLUTION
            (* DESCRIBES THE REQUIRED MAXIMUM VALUE OF THE LEAST
               SIGNIFICANT BIT FOR THE DATA IN UNITS SPECIFIED IN
               THE UNITS ATTRIBUTE. *).
   APPLICABLE ELEMENT_TYPE:  DATA.
   VALUE: NUMERIC.

ATTRIBUTE:  TYPE
            (* THE TYPE FOR A DATA ITEM WHICH IS EITHER
               REFERENCED ON AN R_NET OR SUBNET OR IS USED IN A
               BETA OR GAMMA SIMULATION. *).
   APPLICABLE ELEMENT_TYPE:  DATA.
   VALUE: REAL.
   VALUE: ENUMERATION
         (* THE DATA ITEM CAN ASSUME ONLY CERTAIN VALUES
            WHICH ARE NAMES.  THE ALLOWED VALUES FOR THE DATA
            ITEM ARE SPECIFIED IN THE RANGE ATTRIBUTE. *).
   VALUE: BOOLEAN.
   VALUE: INTEGER.

ATTRIBUTE:  UNITS
            (* THE ENGINEERING UNITS OF THE VALUE OF A DATA ITEM
               OR THE UNITS IN WHICH THE MAXIMUM_TIME AND/OR
               MINIMUM_TIME FOR A VALIDATION_PATH ARE
               SPECIFIED. *).
   APPLICABLE ELEMENT_TYPE:  DATA
                             VALIDATION_PATH.
   VALUE: NAMED
         (* FOR INDIVIDUAL PROJECTS IT MAY BE DESIRABLE TO
            RESTRICT THE UNITS WHICH CAN BE USED.  IN THAT CASE,
            NAMED SHOULD BE REPLACED BY THE SPECIFIC LEGAL
            VALUE NAMES. *).
```

```
ATTRIBUTE: USE
             (* QUALIFIES THE USE OF A DATA ITEM IN
                SIMULATION. *).
   APPLICABLE ELEMENT_TYPE:  DATA.
   VALUE: BETA
          (* THE DATA ITEM IS TO BE USED IN FUNCTIONAL
             SIMULATIONS ONLY. *).
   VALUE: GAMMA
          (* THE DATA ITEM IS TO APPEAR IN ANALYTIC
             SIMULATIONS ONLY. *).
   VALUE: BOTH
          (* THE DATA ITEM IS TO BE USED IN BOTH FUNCTIONAL AND
             ANALYTIC SIMULATIONS. *).
```

3.2 ALPHA SEGMENT

The basic processing steps in the description of a set of functional requirements are embodied in the RSL element type ALPHA. The neutral name "ALPHA" is used in order to avoid any implication of a software implementation function (such as interrupt handling). ALPHAs are data processing system requirements which may need to be implemented in several parts of the data processing system for the sake of reliability, because of the need for separability, or due to the prior existence of code.

3.2.1 Executable Descriptions

Within the ALPHA the actual requirement is represented for simulation purposes with an "executable description", called a BETA or GAMMA. The alternatives under investigation might be gross-level functional models of an ALPHA being evaluated with a functional simulation; these functional models are called BETAs. The alternatives might also involve detailed analytic models or algorithms in an analytic "test-bed" being evaluated with an analytic emulation; these latter models are called GAMMAs.

The BETA and GAMMA executable descriptions are written in the PDL 2 or PASCAL[1] language. The executable descriptions look like PASCAL procedures with omission of the procedure heading and with augmenting statements for that part of the requirement representing access to data hierarchies. All of the normal PASCAL programming features and facilities are available, except that all data that is not strictly local to a BETA or GAMMA must be declared via the RSL Data Segment (see Section 3.1). The REVS Simulation Generation function processes these executable descriptions to produce standard PASCAL procedures for incorporation into the BETA or GAMMA simulation.

3.2.2 Referencing Data

Communication between BETAs or GAMMAs of several ALPHAs during simulation is via the DATA described in the Data Segment (see Section 3.1). No

[1]The Texas Instruments Process Design Language (PDL 2) [1] is a PASCAL-based language supported on the TI ASC machines at the ARC and at NRL. PDL 2 extends standard PASCAL [2]. On CDC installations of REVS, only standard PASCAL will be available.

direct communication outside these defined DATA is allowed, or the executable code in the simulation would rapidly grow apart from the specified requirements. Since a simulation will either be a beta or a gamma, communication between a BETA and a GAMMA is not meaningful.

As described in the Data Segment, an ALPHA INPUTS and OUTPUTS DATA specifying the DATA which is used or generated by the ALPHA. These relationships reflect the processing required of the ALPHA and thus, to ensure that the simulations reflect the requirements, remain valid for both BETAs and GAMMAs. This means, for example, that if an ALPHA INPUTS DATA A and B, then A and B are used in both the BETA and GAMMA executable descriptions. In many cases it may be necessary in a functional simulation for the BETA to use a representation of A and B rather than the actual DATA items. This is accomplished by using a data hierarchy. Another DATA item, say DATA C, is defined which INCLUDES DATA A and B and which has USE BETA and an appropriate TYPE. It is then specified that the ALPHA INPUTS DATA C rather than DATA A and B. This means the same thing in a requirements sense as inputting A and B (since obtaining DATA C obtains DATA A and B), but in a functional simulation DATA A and B will not actually exist as variables but will be represented by a single item C.

DATA specified in RSL are represented in simulations as variables of the types assigned by the RSL attribute TYPE. Thus, reference to DATA from within a BETA or GAMMA is via the RSL name of the element consistent with the ordinary PASCAL reference conventions.

3.2.3 Accessing Files

FILEs consist of multiple instances of their constituent parts. In the executable code of the BETA and GAMMA, the analyst must be able to specify clearly which instance he wishes to deal with. Further, he must be able to create and destroy instances in a FILE. To accomplish these manipulations, special operators have been made available to be used in writing the BETAs and GAMMAs. These operators are processed by the Simulation Generation function to generate executable PASCAL code. These extensions are described below; their syntax and more details are given in Section 7.1.1.

The four special operators are CREATE, SELECT, FOR EACH, and DESTROY. The CREATE and DESTROY statements both specify a FILE name.

The CREATE statement designates that a new instance (record) is to be added to the FILE. The DESTROY statement designates that the currently SELECTed record for the FILE is to be destroyed. The SELECT involves the selection of desired instances in a FILE by the application of a subsetting condition, and further selection within this subset by a FIRST or NEXT criterion to uniquely identify an instance.

To explain the operation of the FILE SELECT, one may visualize the FILE as an ordered assemblage with a pointer which may be moved. The SELECT statement first causes repositioning of the pointer: to the first instance if the statement is a SELECT FIRST; or to the one following the current pointer position for a SELECT NEXT. The condition (selection criterion) is then evaluated using the DATA CONTAINED in the instance designated by the pointer. If the expression evaluates to TRUE, the SELECT has found the desired instance. Otherwise the pointer is moved to the next instance and the process repeated. If the condition is omitted, an instance will be SELECTed by positioning of the pointer only.

The search does not go end-around, it terminates when the last instance, as defined by the FILE ordering, is reached. The predefined LOCAL DATA item RECORD_FOUND is set to TRUE if an instance is SELECTed and is set to FALSE if an instance is not found.

After a particular record of a FILE has been SELECTed, all references to DATA CONTAINED in the FILE are assumed to refer to the instance of that DATA in the SELECTed record. Of course, another SELECTion on the FILE will change the instance that is assumed for all references. The SELECTed record in a FILE remains SELECTed until a new SELECTion is made, even though the processing flow may have passed from one ALPHA to another. The SELECTion is local to an R_NET; no FILE instances are SELECTed on the invocation of an R_NET.

The CREATE and DESTROY accomplish implicit SELECTion. After a CREATE, the newly created record is SELECTed. The DESTROY destroys the currently SELECTed record and does not SELECT another record.

The final FILE operator, the FOR EACH, allows the application of processing repeatedly to several instances in a FILE. The FOR EACH specifies a condition and the FILE name, a block of PASCAL code to be executed for records meeting the criterion, and an ENDFOREACH symbol. The embedded PASCAL code will be executed for each record meeting the selection criterion.

The FOR EACH searches through the FILE from first to last just as if a SELECT FIRST followed by SELECT NEXTs had been written. If no instances are found the embedded code is never executed. Since the FOR EACH cycles through all records in the FILE, there is no instance SELECTed after completion of the FOR EACH. FOR EACHs may be nested without affecting the operation of any of the FOR EACHs.

3.2.4 Accessing Entities

ENTITY_CLASSes, like FILEs, consist of multiple instances. Instances to be processed must be clearly specified. This is accomplished by SELECTion and FOR EACH operations on the Requirements Networks and is described in Section 3.3.

3.2.5 Operations on Entities and Messages

The RSL relationships CREATES, DESTROYS and SETS are between ALPHAs and ENTITY_CLASSes and ENTITY_TYPEs. They indicate that an ALPHA determines the existence of an instance in an ENTITY_CLASS (CREATES and DESTROYS) and its specific ENTITY_TYPE (SETS). The relationship FORMS between an ALPHA and a MESSAGE indicates that the ALPHA designates that the MESSAGE will be PASSED by the appropriate OUTPUT_INTERFACE when the interface is encountered subsequently on the net.

To ensure consistent representation of the requirements in a simulation, code representing these actions is automatically inserted in an ALPHA's executable description (BETA or GAMMA) when a simulation is generated. CREATES and SETS are performed immediately in the BETA or GAMMA -- the CREATES being performed first. DESTROYS and FORMS are performed immediately before exiting a BETA or GAMMA after any user specified code.

3.2.6 Summary of Alpha Segment Concepts

The element type and attributes defined below constitute the Alpha Segment:

ELEMENT TYPE

```
ELEMENT_TYPE:  ALPHA
                  (* A BASIC PROCESSING STEP IN THE FUNCTIONAL
                     REQUIREMENTS. *).
   STRUCTURE APPLICABILITY:  NET.
```

ATTRIBUTES

```
ATTRIBUTE:  BETA
              (* THE PROCEDURAL CODE (PASCAL) FOR FUNCTIONALLY
                 MODELING THE PROCESSING STEP.  THE CODE IS NOT
                 PROCESSED BY THE RSL TRANSLATOR BUT IS PROCESSED
                 BY THE SIMULATION GENERATION FUNCTION AND THE
                 COMPILER.  A BETA MAY USE THE SPECIAL CREATE,
                 DESTROY, SELECT AND FOR EACH OPERATIONS ON
                 FILES. *).
   APPLICABLE ELEMENT_TYPE:  ALPHA.
   VALUE: TEXT.

ATTRIBUTE:  GAMMA
              (* THE PROCEDURAL CODE (PASCAL) FOR ANALYTICALLY
                 MODELING A PROCESSING STEP.  THE CODE IS NOT
                 PROCESSED BY THE RSL TRANSLATOR BUT IS PROCESSED BY
                 THE SIMULATION GENERATION FUNCTION AND THE
                 COMPILER.  A GAMMA MAY USE THE SPECIAL CREATE,
                 DESTROY, SELECT AND FOR EACH OPERATIONS ON
                 FILES. *).
   APPLICABLE ELEMENT_TYPE:  ALPHA.
   VALUE: TEXT.
```

3.3 REQUIREMENTS NETWORK SEGMENT

The specification of the flow of processing steps in RSL is called a Requirements Network or R_NET. Each R_NET details the response of the system to particular stimuli. It specifies the sequence of ALPHAs to be followed to generate changes in system state and responses to the environment. When all of the required steps are completed, the R_NET processing terminates. This sequence of ALPHAs is specified by giving a graph model of the sequence in a structure declaration associated with the R_NET.

3.3.1 Top-Down Flow Specification

RSL has been designed to allow the top-down development of R_NETs. The engineer may first specify system responses in terms of a few ALPHAs which state the general idea of the processing necessary. At a later time, each of these ALPHAs may be expanded in terms of a flow graph of lower level ALPHAs. This expansion process changes the original ALPHA to a SUBNET, a named processing net which exists as a part of a higher-level net. Therefore, each SUBNET is a single entry-single exit flow; i.e., there is one and only one point in the processing flow that RETURNs to the higher level net. All paths either rejoin and RETURN at this point or TERMINATE processing (alternately ending at an OUTPUT-INTERFACE).

An RSL SUBNET is analogous to a macro in a programming language. The SUBNET is treated as though the flow path(s) in that SUBNET were physically inserted into the higher level flow path. This means that the data available to ALPHAs in the R_NET and to ALPHAs in any level SUBNET are identical.

3.3.2 Enablement

The relationship ENABLES provides the mechanism for defining the stimuli which start processing on R_NETs. An R_NET, which is the object of the relationship, is ENABLED for processing by the element which is the subject of the relationship. Two element types are legal as subjects of ENABLES, corresponding to two distinct situations which provide the stimulus for processing.

The first situation is enablement by a stimulus to an R_NET somewhere in the data processing subsystem. The element type EVENT has been defined for this case. An EVENT must be specified in the structure associated with an R_NET. The object R_NET is ENABLED when control passes through the EVENT node on the R_NET which contains it. An R_NET may be enabled by an EVENT on its own structure. Also, the action of an EVENT may be postponed by use of the DELAYS relation. The subject of the relation is a DATA item which specifies how long ENABLEment is to be DELAYED when the object EVENT is triggered. DELAYS and self-ENABLEment allow the specification of functions which are to be performed periodically. An R_NET may be ENABLED by more than one EVENT. The passage of control through each EVENT will result in a separate ENABLEment of the object R_NET. An EVENT may ENABLE more than one R_NET.

The other ENABLEment situation concerns stimuli from outside the data processing subsystem. In this case the subject element of ENABLES is an INPUT_INTERFACE. An INPUT_INTERFACE, which is defined more completely in the Data Segment portion of this document (Section 3.1), provides communication between the data processing system and some other SUBSYSTEM. Data present at the INPUT_INTERFACE is defined to be the condition which causes ENABLEment of an R_NET. An R_NET may be ENABLED by only one INPUT_INTERFACE; an EVENT and an INPUT_INTERFACE may not ENABLE the same R_NET; and an INPUT_INTERFACE may ENABLE only one R_NET.

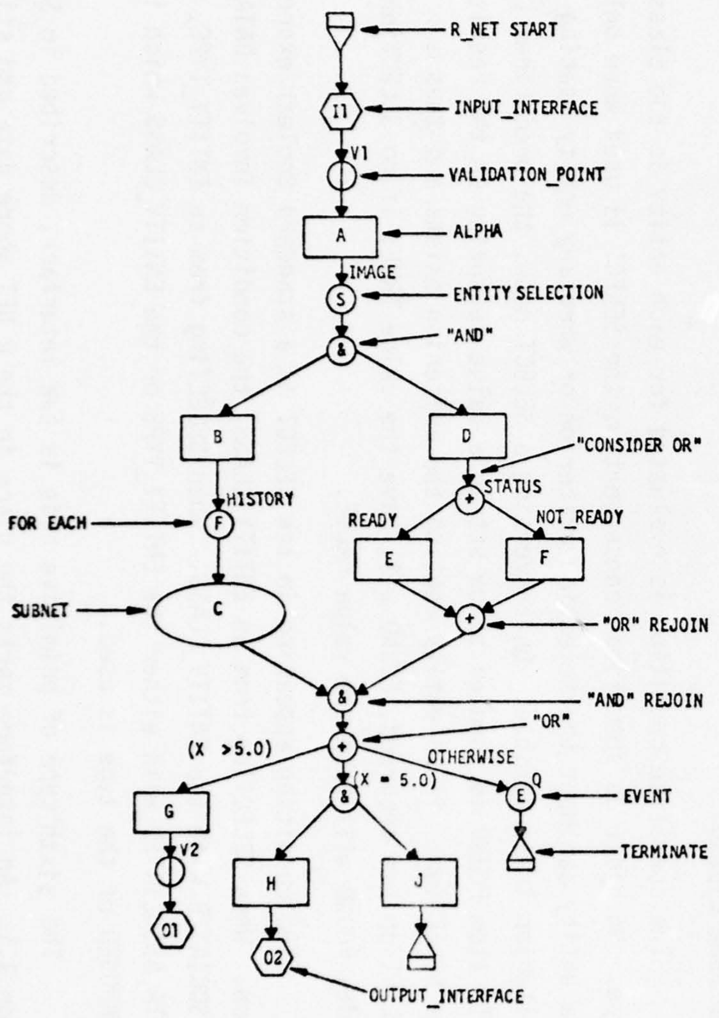
3.3.3 Structure

The flow structure of an R_NET consists of two classes of nodes, primitive and complex, and the arcs which join them. Figure 3-2 shows each of the RSL structure nodes; also illustrated are the corresponding graphics symbols. As can be seen in this figure, the primitive nodes are single entry and single exit; they are the ones that specify the processing steps and related ideas. The complex nodes are multiple entry or multiple exit and express information about the sequencing of the primitive nodes.

Five types of primitive nodes may be placed at any point on the structure except at the ends of the processing paths. They are:

1) ALPHAs (defined in Section 3.2 of this document) - the primitive processing steps.

2) SUBNETs (defined above) - lower level flow structures.



```
R_NET:  SAMPLE.
STRUCTURE:
    INPUT_INTERFACE   I1
    VALIDATION_POINT   V1
    ALPHA A
    SELECT ENTITY_CLASS IMAGE SUCH THAT (Y < Z)
    DO
        ALPHA B
        FOR EACH FILE HISTORY RECORD
            DO SUBNET C END
    AND
        ALPHA D
        CONSIDER DATA STATUS
        IF (READY)
            ALPHA E
        OR (NOT_READY)
            ALPHA F
        END
    END
    IF (X > 5.0)
        ALPHA G
        VALIDATION_POINT V2
        OUTPUT_INTERFACE 01
    OR (X = 5.0)
        DO
            ALPHA H
            OUTPUT_INTERFACE 02
        AND ALPHA J
            ALPHA J
            TERMINATE
    OTHERWISE
        EVENT Q
        TERMINATE
    END
END.
```

Figure 3-2 Sample R_NET Structure in RSL and in Graphical Form

3) VALIDATION_POINTs (see Section 3.4) - data collection points for performance measurement.

4) EVENTs (defined above) - nodes which ENABLE other R_NETs.

5) SELECTs (defined below) - nodes which identify entity instances.

The SELECT node identifies an entity instance to which the processing subsequently specified on the R_NET is to be applied. The SELECT designates an ENTITY_CLASS or ENTITY_TYPE from which an entity instance is to be chosen and a condition (selection criterion) involving DATA ASSOCIATED with the entity. It specifies that the entity instance meeting the criterion is to be used in subsequent processing. After an entity instance is SELECTed, it remains SELECTed until either the R_NET terminates or another SELECT is performed on the same ENTITY_CLASS or on any ENTITY_TYPE which COMPOSES the class containing the entity instance. After a SELECT, there is at most one entity selected from an ENTITY_CLASS even though the SELECT considered only one ENTITY_TYPE composing the class (i.e., a SELECTion on one ENTITY_TYPE negates any previous SELECTion on the same type or on a different type of the same class).

The specified condition is evaluated for each entity in the class or type. No order is specified; consequently, the SELECT is used when only one entity can meet the selection criterion or when any entity meeting the criterion is acceptable. On traversing a SELECT node, the predefined LOCAL DATA item FOUND is assumed to be set to a value depending on the result of the SELECTion: if an entity meeting the criterion exists and thus a SELECTion has occurred, FOUND will have the value TRUE; if no SELECTion is made, FOUND will have the value FALSE.

The condition appearing in the SELECT is a standard Boolean expression. When SELECTing from an ENTITY_CLASS, the condition involves DATA ASSOCIATED with the ENTITY_CLASS. When SELECTing from an ENTITY_TYPE, DATA ASSOCIATED with either the ENTITY_TYPE or the ENTITY_CLASS which is COMPOSED of the type is used.

The sixth type of primitive node is the interface, described in Section 3.1. An interface marks the place in the R_NET where data and stimuli are introduced from or communicated to the outside world. Interfaces are restricted to being at the beginning of an R_NET (for an INPUT_INTERFACE) or at the ends of R_NET paths (for an OUTPUT_INTERFACE).

The seventh and eighth types of primitive nodes also signal the end of a flow path; naturally, they may be placed only at the end of the structure declaration or at the end of a flow path within the structure. A TERMINATE node means that all processing on that flow path is ended. It is not specified if an R_NET path finishes at an OUTPUT_INTERFACE. The RETURN* node means that the end of processing on the main path of a SUBNET structure has been reached and that the flow continues on the higher level structure which references the SUBNET.

The first of the three complex nodes, the FOR EACH node, specifies a repetitive data structure (an ENTITY_CLASS, an ENTITY_TYPE, or a FILE), an ALPHA or SUBNET, and an optional condition involving DATA related to the structure. The SUBNET or ALPHA is performed once for each instance in the structure that meets the condition.** The order of evaluation is not specified, so the different invocations of the ALPHA or SUBNET may be assumed to be done in a "don't care sequence". Thus, the FOR EACH node may be looked upon as an iterative operation (without sequential implications) over the repetitive structure with the application of the condition as a subsetting criterion. (It can also be considered as an AND structure, the second complex node to be discussed next, with the number of branches known only at the time of invocation of the FOR EACH.) As stated above the condition is optional; if omitted, the ALPHA or SUBNET is performed for all instances in the structure.

Again, the condition is a standard Boolean expression. When the repetitive structure is an ENTITY_CLASS, the criterion involves DATA ASSOCIATED with the ENTITY_CLASS. For an ENTITY_TYPE, the DATA used in the condition may be ASSOCIATED with either the ENTITY_TYPE or the ENTITY_CLASS which the type COMPOSES. In the case of a FILE, the criterion references DATA CONTAINED in the FILE.

*The graphic symbol for a RETURN is a Δ , that for a SUBNET start node is a ∇.

**In the graphical form of an R_NET structure, the FOR EACH node is represented by two symbols: one to designate the FOR EACH and another for the ALPHA or SUBNET in the scope of the FOR EACH (see Figure 3-2). The ALPHA or SUBNET symbol always immediately follows the FOR EACH symbol.

Since the FOR EACH on an ENTITY_CLASS or ENTITY_TYPE evaluates the condition for all entities in the specified class or type and functions as a SELECTion for the ALPHA or SUBNET, the FOR EACH node negates the result of any previous SELECT node for the pertinent ENTITY_CLASS or any ENTITY_TYPE which COMPOSES the class -- i.e., after completion of a FOR EACH, there is no entity SELECTed within the ENTITY_CLASS which is the subject of the FOR EACH or which is COMPOSED of the subject ENTITY_TYPE. Similarly, after a FOR EACH on a FILE, no instance is identified as SELECTed for the FILE. (FILE SELECTions also occur within ALPHAs and are discussed in Section 3.2).

The **second** type of complex node is the AND node. This node has one entry arc and several exit arcs. The meaning is that all of the exit arcs are followed but the order of processing of the paths is not relevant in a requirements sense -- in fact, the arcs may be followed in parallel as long as the meaning of the data is maintained. Thus the parallel branches of the flow graph following this node are followed without regard to order; they form a "don't care set". The requirements analyst may, by using AND nodes, specify that no order need be imposed on a group of ALPHAs.

There are two classes of AND structures. In the first class, the parallel branches do not rejoin; each of them ends with a TERMINATE or an OUTPUT_INTERFACE. The branches for this split-off AND are independent of each other and have no mutual timing constraints. The other class is the rejoining AND. In this structure, there is a virtual AND node with multiple entries and a single exit which collects all of the parallel branches. This virtual node will not be passed until all of the branches have been processed. In this sense, it acts as a synchronizing node. There is no syntactic distinction between these two classes of AND structures; the composition of the branches determines the class. Either all of the branches terminate or none of them do; a mixed case is not allowed.

The **third** and final type of complex node is the OR node. It also has one entry and several exits, but only one of the exit arcs is followed. The choice is made based on a condition associated with each arc. The OR node and its accompanying structure represent the folding together of several paths of processing with structures which are identical to a point, but differ based on some choice criterion. If the paths differ from that point on, the OR is a split-off OR. If the paths are again identical at some later point, the branches may be collected at a virtual OR node with several entries and one exit. This structure is a rejoining OR. Again, mixed rejoining and split-off OR branches are not allowed.

There are two types of OR nodes: the basic OR node and the CONSIDER OR node. For the basic OR node the condition on each exit arc of the node is a standard Boolean expression which may involve DATA elements and constants. This condition is evaluated when the OR node is reached. With general conditions such as this, it is important to assure that the choice of branch is well-defined, since more than one condition may be true, or possibly all of the conditions may be false. For the former case, the analyst may specify an ordinal for each condition. The ordinal gives the position of that condition in an evaluation order. When the OR node is reached, the conditions are evaluated in the specified order, and the first condition which evaluates to TRUE specifies the branch to be followed. If the ordinals are not given, the lexical ordering of the conditions (i.e., the order in which the conditions are entered) is taken as the order of evaluation. To prevent problems if all conditions are false, an OTHERWISE clause is required for the basic OR node. This clause specifies a branch which is followed only if none of the conditions are true.

The second type of OR node, the CONSIDER OR node, allows branching on the value of DATA which has TYPE ENUMERATION (see Section 3.1.7), or branching on the ENTITY_TYPE of the currently SELECTed entity of a particular ENTITY_CLASS. With each branch of the CONSIDER OR node is associated a criterion. Each criterion consists of a single name or a list of names separated by the word OR.

DATA with TYPE ENUMERATION have values which are names. The criteria specify which branch is to be taken based on the value of the DATA under CONSIDERation. The legal value names of the DATA are specified in the RANGE attribute of DATA (see Section 3.1.7). All of the legal value names must appear once and only once in the branching criteria of a CONSIDER OR. In order to prevent confusion of the value names with DATA names which may occur in the basic OR node branch conditions, DATA items with TYPE ENUMERATION may be referenced on a structure only in CONSIDER OR nodes.

The branch criteria when an ENTITY_CLASS is being CONSIDERed contain the names of the ENTITY_TYPEs which COMPOSE the ENTITY_CLASS. Each type composing the class is referenced once and only once in the branching criteria.

Since the criteria for a CONSIDER OR must be exhaustive, an OTHERWISE branch is not allowed.

3.3.4 Summary of Requirements Network Segment Concepts

The element types and relationships defined below, along with the structure declaration, constitute the Requirements Network Segment.

ELEMENT TYPES

```
ELEMENT_TYPE:  EVENT
               (* AN IDENTIFIED POINT IN THE SEQUENCE OF
                  PROCESSING SPECIFIED BY ONE OR MORE R_NETS (OR
                  SUBNETS) WHICH CAUSES THE ENABLEMENT OF AN
                  R_NET.  AN EVENT MAY BE USED TO SPECIFY A
                  VALIDATION_PATH. *).
   STRUCTURE APPLICABILITY:  NET.
   STRUCTURE APPLICABILITY:  PATH.
ELEMENT_TYPE:  R_NET
               (* THE ORDER OF LOGICAL PROCESSING THAT MUST BE
                  PERFORMED BY THE DATA PROCESSING SUBSYSTEM IN
                  RESPONSE TO EXTERNAL OR INTERNAL STIMULI.  THE
                  PROCESSING STEPS ARE ALPHAS OR SUBNETS WHICH
                  MAY BE EXPANDED TO LOWER LEVELS OF DETAIL.  IN
                  ADDITION TO PROCESSING STEPS, THE R_NET
                  STRUCTURE MAY CONTAIN INTERFACES, EVENTS,
                  VALIDATION_POINTS, ANDS, ORS, SELECTS, AND FOR
                  EACH NODES; IT MUST BE ENABLED AND
                  TERMINATED. *).
ELEMENT_TYPE:  SUBNET
               (* A SEQUENCE OF LOGICAL PROCESSING STEPS THAT
                  MUST BE PERFORMED TO ACCOMPLISH THE
                  REQUIREMENTS OF THE NEXT HIGHER NETWORK
                  (SUBNET OR R_NET). *).
   STRUCTURE APPLICABILITY:  NET.
```

RELATIONSHIPS

```
RELATIONSHIP:  DELAYS
               (* THE ENABLEMENT OF R_NETS BY THE EVENT IS
                  POSTPONED FOR THE AMOUNT OF TIME SPECIFIED IN
                  THE DATA.  ONLY ONE DATA MAY DELAY AN EVENT;
                  THIS DATA MUST NOT INCLUDE OTHER DATA.  FOR
                  SIMULATION PURPOSES, THE VALUE OF THIS DATA
                  MUST BE IN UNITS OF SECONDS. *).
   COMPLEMENTARY RELATIONSHIP:  DELAYED ("BY").
   SUBJECT ELEMENT_TYPE:  DATA.
   OBJECT ELEMENT_TYPE:  EVENT.

RELATIONSHIP:  ENABLES
               (* INDICATES THAT WHEN THE PROCESSING CONTROL FLOW
                  PASSES THROUGH THE EVENT ON AN R_NET, OR WHEN
                  DATA IS AVAILABLE AT THE INPUT_INTERFACE, THE
                  FUNCTIONAL PROCESSING SPECIFIED BY THE R_NET
                  CAN BE BEGUN.  AN R_NET MUST BE ENABLED AND CAN
                  BE ENABLED EITHER BY ONE AND ONLY ONE
                  INPUT_INTERFACE OR BY ONE OR MORE EVENTS. *).
   COMPLEMENTARY RELATIONSHIP:  ENABLED ("BY").
   SUBJECT ELEMENT_TYPE:  EVENT
                          INPUT_INTERFACE.
   OBJECT ELEMENT_TYPE:  R_NET.
```

3.4 VALIDATION SEGMENT

The R_NETs, ALPHAs, and DATA concepts of RSL are used to describe the functional requirements for processing. The concepts comprising the Validation Segment of RSL support the definition of the performance requirements to be met by the real-time software.

Two types of performance requirements are expressed in RSL: response time requirements for a path of processing, and accuracy and more complex timing relationships. The response time requirements are stated separately because they apply to single paths and are a common type of requirement. In both cases, specific paths within R_NETs are identified.

3.4.1 Validation Points

R_NET paths are identified using VALIDATION_POINTs. The points appear as nodes on R_NET or SUBNET structures and are somewhat analogous to test points in a piece of electronic hardware. Conceptually, the VALIDATION_POINT transfers information to a recording system which records the relevant information for post-test analysis to determine whether accuracy and timing performance requirements have been satisfied by the process. Since the system under test is a data processing system, the information transferred consists of DATA and FILEs. A VALIDATION_POINT RECORDS DATA and FILEs; these DATA and FILEs are those necessary to describe and/or test a performance requirement.

For DATA ASSOCIATED with ENTITY_CLASSes and ENTITY_TYPEs, only DATA ASSOCIATED with the currently SELECTed entity may be RECORDED. To RECORD DATA for more than one class, the VALIDATION_POINT is placed in the scope of a FOR EACH on the ENTITY_CLASS or one of its ENTITY_TYPEs.

For RECORDED DATA which is CONTAINED in a FILE, the DATA values in the currently SELECTed (by an ALPHA or a FOR EACH on a net) record in the FILE are RECORDED. If the FILE is also RECORDED, then the specified DATA is RECORDED from each record of the FILE.

There is no explicit output from a VALIDATION_POINT. As stated above, a model for the use of the DATA is that they are recorded in a data file to be used in a post-processing mode to determine whether the performance requirements have been met. Thus for each traversal of a VALIDATION_POINT, a recording is generated which contains the DATA and FILEs RECORDED by the VALIDATION_POINT.

3.4.2 Validation Paths

The individual paths through an R_NET which are of interest in describing performance requirements are stated in RSL as VALIDATION_PATHs. The path structure of a VALIDATION_PATH is primarily a sequence of VALIDATION_POINTs on a single R_NET. A path thus described must correspond to a route through an R_NET; it is illegal, for instance, to specify that a path exists between VALIDATION_POINTs which are on parallel branches of an AND or OR structure. If all paths between two nodes are to be included in the requirements, then the requirements engineer need merely refrain from noting any VALIDATION_POINTs between the nodes. Therefore, if all paths through an R_NET are included, just the starting and ending VALIDATION_POINTs need to be noted. For example, consider the R_NET structure shown in Figure 3-3. If both paths through the net are to be included, then a VALIDATION_PATH may be defined using only VALIDATION_POINTs V1 and V3. Conversely, if only one path through a multiple path R_NET is to be considered, sufficient VALIDATION_POINTs must be specified so that the desired path is uniquely identified. In the example, VALIDATION_POINT V2 would be included in the VALIDATION_PATH definition in order to designate the path through ALPHAs A, C and D.

In many instances, timing and accuracy requirements span more than one R_NET. RSL provides a mechanism to extend the VALIDATION_PATH to cover more than one R_NET, if an EVENT on the first R_NET ENABLES the second R_NET. EVENTs are then placed on PATHs so that the flow of processing to the R_NETs which are ENABLED by the EVENT may be followed. Thus the PATH structure of a VALIDATION_PATH contains VALIDATION_POINTs and EVENTs as nodes.

For performance requirements which apply to processing performed by more than one path in an R_NET in multiple executions or by paths on different R_NETs, RSL allows the definition of the VALIDATION_PATHs separately and the definition of a relationship between the performance requirements and the VALIDATION_PATHs as explained in Section 3.4.4.

3.4.3 Stimulus-Response Timing Requirements

Timing requirements can be imposed along a VALIDATION_PATH through use of the attributes MINIMUM_TIME and MAXIMUM_TIME. The points for starting and stopping the stimulus-response timing measurement are the

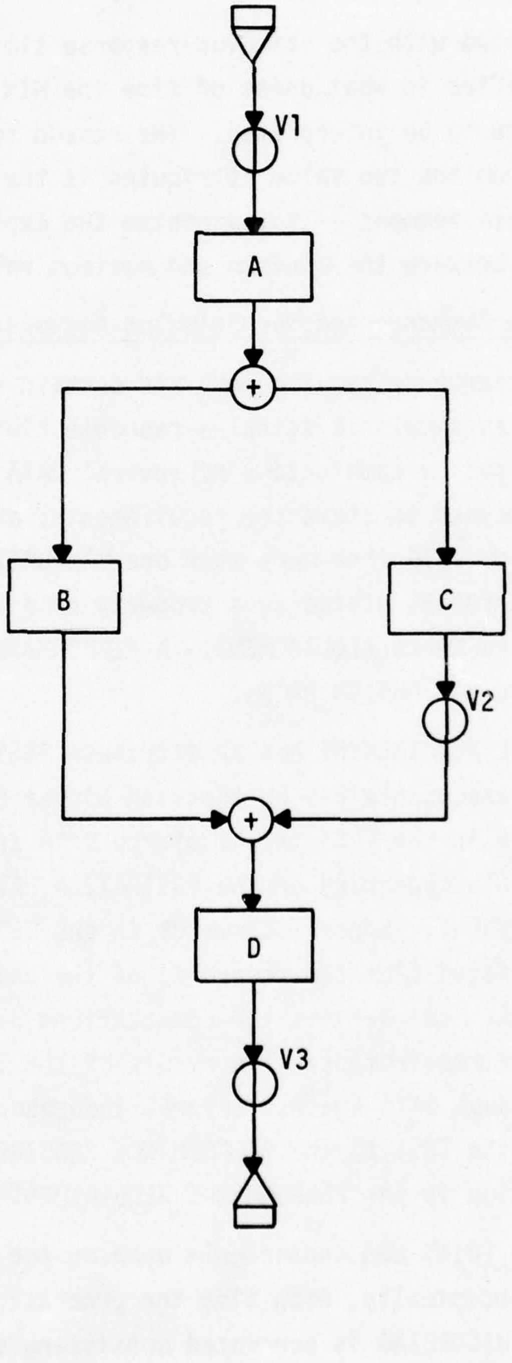


Figure 3-3 Illustration of Validation Paths

start and termination of the R_NET path identified by the PATH structure of the VALIDATION_PATH. If the VALIDATION_PATH specifies multiple paths through the R_NET, the timing requirements are interpreted to mean that all of these R_NET paths must meet the timing constraints.

Also associated with the stimulus-response timing is the attribute UNITS, which specifies in what units of time the MINIMUM_TIME and MAXIMUM_TIME attributes are to be interpreted. The reason for separation of the UNITs attribute from the two value attributes is the same in this segment as it is in the Data Segment -- to guarantee the explicitness and consistency of units between the minimum and maximum values.

3.4.4 Analytic Performance and Non-Stimulus-Response-Timing Requirements

Analytic performance requirements and certain timing requirements can not be stated as simply as stimulus-response timing requirements. Two reasons make this so: a combination of several DATA and a complex transformation may be needed to state the requirements; and the requirements may be a function of DATA from more than one VALIDATION_PATH. Therefore, the requirement cannot be stated as a property of a VALIDATION_PATH but is stated as a PERFORMANCE_REQUIREMENT. A PERFORMANCE_REQUIREMENT CONSTRAINS one or more VALIDATION_PATHs.

A PERFORMANCE_REQUIREMENT has an attribute TEST which defines the requirement as an executable PASCAL function with a Boolean value. The information available in the TEST is all of the DATA and FILEs RECORDED by the VALIDATION_POINTs appearing on the VALIDATION_PATHs CONSTRAINED by the PERFORMANCE_REQUIREMENT. Special commands in the TEST identify how the data is to be extracted from the recording of the validation point information, and the PASCAL code defines the computations necessary to test the satisfaction of the requirement. The result of the TEST, which may examine a number of individual DATA against several independent criteria, is the value assigned by the TEST to the PERFORMANCE_REQUIREMENT function. The name of this function is the PERFORMANCE_REQUIREMENT name.

A VALIDATION_POINT can appear only once on the nets but can appear on many paths. Conceptually, each time the processing reaches a VALIDATION_POINT on a net, a RECORDING is generated consisting of all DATA RECORDED by the VALIDATION_POINT. A VALIDATION_POINT can RECORD a FILE; in which case DATA is extracted for each record in the FILE. The same DATA and FILES may be RECORDED by many VALIDATION_POINTs.

Thus, DATA referenced in a TEST must be uniquely identified by VALIDATION_POINT, by RECORDING, and by record in a FILE and a FILE must be identified by VALIDATION_POINT and RECORDING. The approach adopted to establish uniqueness is analogous to that used in the remainder of RSL for DATA and FILEs ASSOCIATED with entities and for DATA CONTAINED in FILEs. The main difference is that all DATA and FILE names appearing in the TEST are prefixed by the name of the VALIDATION_POINT which RECORDED the DATA or FILE to be used. The two names are separated by a decimal point. (Thus, to refer to DATA (or FILE) A RECORDED by VALIDATION_POINT V1, the identifier V1.A is used in the TEST.)

The special operators available in TESTs for identifying a particular RECORDING are the RETRIEVE and FOR EACH. These have the identical meaning as the SELECT and FOR EACH on FILEs written in BETAs and GAMMAs described in Section 3.2.3. After a RETRIEVE operation, the Boolean variable RECORDING_FOUND will have the value TRUE if a RECORDING which meets the retrieval criterion was located; otherwise, RECORDING_FOUND will have the value FALSE. The RECORDING RETRIEVEd remains available until the next RETRIEVE or FOR EACH is encountered on the same VALIDATION_POINT. The FOR EACH in a TEST has the same meaning as the FOR EACH statement in BETA/GAMMA code; the code encompassed by the DO and ENDFOREACH in the FOR EACH operation is executed in sequence for each RECORDING meeting the retrieval criterion. A SELECT and FOR EACH on FILEs are also available for writing TESTs and have this same interpretation. The syntax for each of the special TEST operators is presented in Section 7.1.2.

3.4.5 Summary of Validation Segment Concepts

The element types, relationship, and attributes defined below constitute the Validation Segment:

ELEMENT TYPES

```
ELEMENT_TYPE:  PERFORMANCE_REQUIREMENT
               (* AN ANALYTIC PERFORMANCE REQUIREMENT OR
                  NON-STIMULUS-RESPONSE TIMING REQUIREMENT WHICH
                  IS TO BE MET BY THE DATA PROCESSING
                  SUBSYSTEM. *).
```

```
ELEMENT_TYPE:  VALIDATION_PATH
               (* A PATH OF PROCESSING OVER WHICH QUANTITATIVE
                  VALIDATION TESTING WILL BE PERFORMED.  A PATH
                  IS SPECIFIED USING VALIDATION_POINTS AND
                  EVENTS AND MUST CORRESPOND TO A ROUTE THROUGH
                  AN R_NET OR THROUGH R_NETS CONNECTED BY
                  EVENTS. *).

ELEMENT_TYPE:  VALIDATION_POINT
               (* A LOGICAL POINT IN THE PROCESSING SPECIFIED BY
                  AN R_NET OR SUBNET AT WHICH DATA MUST BE
                  OBTAINABLE IN THE IMPLEMENTED SOFTWARE IN ORDER
                  TO VALIDATE THAT THE PERFORMANCE REQUIREMENTS
                  HAVE BEEN FULFILLED. *).
   STRUCTURE APPLICABILITY:  NET.
   STRUCTURE APPLICABILITY:  PATH.
```

RELATIONSHIP

```
RELATIONSHIP:  CONSTRAINS
               (* IDENTIFIES TO WHICH VALIDATION_PATH(S) THE
                  PERFORMANCE_REQUIREMENT APPLIES. *).
   COMPLEMENTARY RELATIONSHIP:  CONSTRAINED ("BY").
   SUBJECT ELEMENT_TYPE:  PERFORMANCE_REQUIREMENT.
   OBJECT ELEMENT_TYPE:  VALIDATION_PATH.
```

ATTRIBUTES

```
ATTRIBUTE:  MAXIMUM_TIME
            (* THE MAXIMUM TIME THAT CAN BE TAKEN TO TRAVERSE THE
               VALIDATION_PATH.  THE TIME IS SPECIFIED IN THE UNITS
               STATED IN THE UNITS ATTRIBUTE. *).
   APPLICABLE ELEMENT_TYPE:  VALIDATION_PATH.
   VALUE: NUMERIC.

ATTRIBUTE:  MINIMUM_TIME
            (* THE MINIMUM TIME THAT CAN BE TAKEN TO TRAVERSE THE
               VALIDATION_PATH.  THE TIME IS SPECIFIED IN THE
               UNITS DESIGNATED BY THE UNITS ATTRIBUTE. *).
   APPLICABLE ELEMENT_TYPE:  VALIDATION_PATH.
   VALUE: NUMERIC.

ATTRIBUTE:  TEST
            (* PROCEDURAL CODE (PASCAL) WHICH DEFINES THE
               COMPUTATIONS NECESSARY TO TEST THE SATISFACTION OF
               A PERFORMANCE_REQUIREMENT USING DATA RECORDED BY
               VALIDATION_POINTS.  THE CODE IS NOT PROCESSED
               BY THE RSL TRANSLATOR BUT IS PROCESSED BY THE
               SIMULATION GENERATION FUNCTION AND THE COMPILER.
               A TEST CONTAINS SPECIAL RETRIEVE AND FOR EACH
               OPERATIONS TO IDENTIFY VALIDATION_POINT RECORDINGS
               AND MAY USE SELECT AND FOR EACH OPERATIONS TO
               ACCESS RECORDED FILES. *).
   APPLICABLE ELEMENT_TYPE:  PERFORMANCE_REQUIREMENT.
   VALUE: TEXT.
```

3.5 MANAGEMENT SEGMENT

Concepts necessary to support the disciplined management of a requirements engineering project are contained in the Management Section of RSL. These concepts are broadly defined and provide a framework of useful information which can be adapted to many types of project management.

3.5.1 Configuration Management

The size of modern software projects dictates that close track be kept of changes to the system at all levels, from requirements to code. To support configuration management, information must be maintained about the nature and author of each change to the system. In RSL, this information can be maintained about each element of the requirements description. Updates to each element in the ASSM provide the change history, including the individual's identity, through the ENTERED_BY attribute; and the ASSM itself provides the current state of the element. The attribute COMPLETENESS (which can be associated with any element) provides a means for the requirements engineer to identify how close his entry is to its final form. If he is merely noting initial ideas, he can give the attribute the value INCOMPLETE. The other attribute values (CHANGEABLE and COMPLETE) indicate increasing finality of the requirements.

3.5.2 Traceability

The output of the software requirements engineering effort should be directly traceable upward to the originating requirements in the system requirements documentation. Data on these originating requirements are given in RSL through the use of the element type ORIGINATING_REQUIREMENT. Elements which are traced from an ORIGINATING_REQUIREMENT are linked to it by the TRACES relationship.

The TRACES relationship may be used to assess the impact of a change to an ORIGINATING_REQUIREMENT. In many cases, though, the ORIGINATING_REQUIREMENTs are interrelated in a hierarchical manner, such that lower level requirements add detail to more global higher level requirements. This hierarchy is reflected by the relationship INCORPORATES between ORIGINATING_REQUIREMENTs.

The traceability downward to different versions of the system is also facilitated by the element type VERSION and the relationship IMPLEMENTS.

Any element of the requirements can thus be identified as being IMPLEMENTED by only a few VERSIONs or by all of them.

Some requirements may need to be implemented precisely in the real-time software in order that the data processing system will function correctly, even though they are not directly traceable to system requirements. The degree to which an element is artificial, therefore, is associated with its degree of traceability, but not perfectly. For this reason ARTIFICIALITY is a separate attribute that can be associated with any element in the stated requirements.

3.5.3 Decisions

Although some ORIGINATING_REQUIREMENTs can be directly allocated to lower levels (and TRACED to those lower levels), many ORIGINATING_REQUIREMENTs must be combined with other information in order to take them to lower levels. This may occur because the ORIGINATING_REQUIREMENT was ambiguous or incomplete, because many alternative interpretations of the requirement may exist and an assumption or choice must be made to proceed, or because a literal interpretation of the ORIGINATING_REQUIREMENT would cause conflict with other ORIGINATING_REQUIREMENTs or with common sense. In each case, the requirements engineer must make some considered decision about how to proceed; the decisions and the considerations (as well as the initial problem) must be documented so that other requirements engineers know why derived (as opposed to allocated) decisions have been made. In addition, an ORIGINATING_REQUIREMENT may change, and documented decisions will need reconsideration; they can be reconsidered far more rapidly and efficiently than undocumented decisions.

The element type DECISION is provided to enable the requirements engineer to note these derived requirements. It is related to one or more ORIGINATING_REQUIREMENTs or other DECISIONs and to the resulting requirements through the relationship TRACES. It has attributes of PROBLEM, ALTERNATIVES, and CHOICE which provide the needed documentation.

3.5.4 Source Material

The element type ORIGINATING_REQUIREMENT does not reflect the origin of the requirement;i.e., whether it appeared in a system specification, an interface specification or some other document. In addition to requirements derived from ORIGINATING_REQUIREMENTs, there may be some which arise from conditions documented in background material. If this background information changes, the requirements must also change; therefore, the background source should appear in the requirements description and should be explicitly traceable to elements which it influences. The element type SOURCE exists to fill these needs. A SOURCE DOCUMENTS elements which depend on material in the SOURCE.

3.5.5 Synonyms

Alternative or shortened names for elements of a requirements description may be entered by defining an element of type SYNONYM. The SYNONYM is linked to the original element by the relationship EQUATES. An original name may be EQUATED to any number of SYNONYMS, but each SYNONYM EQUATES to one and only one element.

3.5.6 Unstructured Information

Additional commentary or descriptive information about an element which does not fit into the context of the relationships or attributes which are applicable to that element may be included in the attribute DESCRIPTION. The value for DESCRIPTION is a text string which has no predefined structure. A requirements engineer may, therefore, include in DESCRIPTION an English language explanation or summary of the element.

In addition to the capability of including unstructured commentary about the normally-structured requirements, RSL provides a means for documenting requirements which do not fit into the patterns established by the standard element types. This type of requirement, for example one which says that the implementation language shall be FORTRAN, may be included as an UNSTRUCTURED_REQUIREMENT in the requirements. Since an UNSTRUCTURED_REQUIREMENT is an element, all of the normal management relationships such as TRACES or EQUATES are applicable to it. This allows the inclusion of the unstructured information in the management process.

3.5.7 Summary of Management Segment Concepts

The element types, relationships, and attributes in the Management Segment are presented below:

ELEMENT TYPES

```
ELEMENT_TYPE:  DECISION
               (* A CHOICE OR INTERPRETATION THAT HAS BEEN MADE
                  IN ORDER TO ESTABLISH FUNCTIONAL AND/OR
                  PERFORMANCE REQUIREMENTS BASED ON ONE OR MORE
                  ORIGINATING_REQUIREMENTS.  THIS MEANS THAT THE
                  LOWER LEVEL REQUIREMENTS ARE A RESULT OF
                  DERIVATION, NOT SIMPLY ALLOCATION. *).

ELEMENT_TYPE:  ORIGINATING_REQUIREMENT
               (* A HIGHER LEVEL REQUIREMENT FROM WHICH LOWER
                  LEVEL REQUIREMENTS (THOSE EXPRESSED IN THE RSL)
                  ARE TRACEABLE. *).

ELEMENT_TYPE:  SOURCE
               (* SOURCE OR AUXILIARY MATERIAL FOR REQUIREMENTS,
                  I.E., ORIGINATING POINT FOR ONE OR MORE
                  ORIGINATING_REQUIREMENTS, DOCUMENTATION OF
                  TRADE-OFF STUDIES, OR BACKGROUND MATERIAL FOR
                  REQUIREMENTS ELEMENTS. *).

ELEMENT_TYPE:  SYNONYM
               (* A SYNONYM IS AN ALTERNATE NAME THAT CAN BE USED
                  IN PLACE OF THE PRIME NAME OF AN ELEMENT.  IT
                  IS USED AS AN ABBREVIATION IN MOST CASES, BUT
                  MAY BE USED FOR OTHER REASONS.  NOTE: IN THE
                  RSL DEFINITIONS OF RELATIONSHIPS AND
                  ATTRIBUTES, "ALL" ALWAYS IMPLIES "ALL EXCEPT
                  SYNONYM". *).

ELEMENT_TYPE:  UNSTRUCTURED_REQUIREMENT
               (* A REQUIREMENT THAT MUST BE PASSED TO THE
                  SOFTWARE DESIGNER BUT THAT DOES NOT FIT INTO
                  THE STRUCTURED FRAMEWORK PROVIDED BY RSL.  THIS
                  ELEMENT MIGHT BE USED BECAUSE THE
                  REQUIREMENT IN QUESTION IS TOO UNCOMMON TO
                  JUSTIFY DEFINITION OF A NEW TYPE OF ELEMENT, A
                  NEW RELATIONSHIP, OR A NEW ATTRIBUTE.  (AN
                  EXAMPLE OF AN UNSTRUCTURED_REQUIREMENT MIGHT
                  BE PRECLUSION OF USING A MULTIPROCESSOR WITH
                  ASSOCIATIVE MEMORY.) *).

ELEMENT_TYPE:  VERSION
               (* THE AGGREGATION OF REQUIREMENTS THAT ARE TO
                  APPLY AS A UNIT TO THE DATA PROCESSING
                  SUBSYSTEM AT A PARTICULAR TIME.  LOOP_1,
                  LOOP_2, ETC., ARE VERSIONS, AS IS AN IOC
                  SYSTEM. *).
```

```
RELATIONSHIP:  DOCUMENTS
               (* THE SOURCE MATERIAL PROVIDES AUXILIARY
                  INFORMATION ABOUT OR IS THE ORIGINATING POINT
                  FOR THE OBJECT ELEMENT. *).
   COMPLEMENTARY RELATIONSHIP:  DOCUMENTED ("BY").
   SUBJECT ELEMENT_TYPE:  SOURCE.
   OBJECT ELEMENT_TYPE:  ALPHA
                         DATA
                         DECISION
                         ENTITY_CLASS
                         ENTITY_TYPE
                         EVENT
                         FILE
                         INPUT_INTERFACE
                         MESSAGE
                         ORIGINATING_REQUIREMENT
                         OUTPUT_INTERFACE
                         PERFORMANCE_REQUIREMENT
                         R_NET
                         SUBNET
                         SUBSYSTEM
                         UNSTRUCTURED_REQUIREMENT
                         VALIDATION_PATH
                         VALIDATION_POINT
                         VERSION.
RELATIONSHIP:  EQUATES ("TO")
               (* DEFINES AN ALTERNATE NAME FOR AN ELEMENT.  THE
                  OBJECT OF EQUATES IS CALLED THE PRIME NAME.
                  THE SUBJECT NAME CAN BE USED FOR INPUT TO THE
                  ASSM, BUT ALL RELATIONSHIPS, ATTRIBUTES, AND
                  STRUCTURES SO DEFINED ARE ACTUALLY
                  CHARACTERISTICS OF THE PRIME NAME. *).
   COMPLEMENTARY RELATIONSHIP:  EQUATED ("TO").
   SUBJECT ELEMENT_TYPE:  SYNONYM.
   OBJECT ELEMENT_TYPE:  ALPHA
                         DATA
                         DECISION
                         ENTITY_CLASS
                         ENTITY_TYPE
                         EVENT
                         FILE
                         INPUT_INTERFACE
                         MESSAGE
                         ORIGINATING_REQUIREMENT
                         OUTPUT_INTERFACE
                         PERFORMANCE_REQUIREMENT
                         R_NET
                         SOURCE
                         SUBNET
                         SUBSYSTEM
                         UNSTRUCTURED_REQUIREMENT
                         VALIDATION_PATH
                         VALIDATION_POINT
                         VERSION.
```

```
RELATIONSHIP:  IMPLEMENTS
               (* DEFINES THE VERSION(S) TO WHICH THE ELEMENT
                  APPLIES. *).
   COMPLEMENTARY RELATIONSHIP:  IMPLEMENTED ("BY").
   SUBJECT ELEMENT_TYPE:  ALPHA
                          DATA
                          DECISION
                          ENTITY_CLASS
                          ENTITY_TYPE
                          EVENT
                          FILE
                          INPUT_INTERFACE
                          MESSAGE
                          ORIGINATING_REQUIREMENT
                          OUTPUT_INTERFACE
                          PERFORMANCE_REQUIREMENT
                          R_NET
                          SUBNET
                          SUBSYSTEM
                          UNSTRUCTURED_REQUIREMENT
                          VALIDATION_PATH
                          VALIDATION_POINT.
   OBJECT ELEMENT_TYPE:  VERSION.

RELATIONSHIP:  INCORPORATES
               (* INDICATES A HIERARCHICAL RELATIONSHIP BETWEEN
                  ORIGINATING_REQUIREMENTS.  THE SCOPE OF THE
                  SUBJECT (HIGHER LEVEL) ORIGINATING_REQUIREMENT
                  INCLUDES THE OBJECT (LOWER LEVEL)
                  ORIGINATING_REQUIREMENT. *).
   COMPLEMENTARY RELATIONSHIP:  INCORPORATED ("IN").
   SUBJECT ELEMENT_TYPE:  ORIGINATING_REQUIREMENT.
   OBJECT ELEMENT_TYPE:  ORIGINATING_REQUIREMENT.
```

```
RELATIONSHIP:  TRACES ("TO")
               (* IDENTIFIES THE ELEMENTS (LOWER LEVEL
                  REQUIREMENTS) TO OR FROM WHICH THE HIGHER
                  LEVEL REQUIREMENT (ORIGINATING_REQUIREMENT OR
                  DECISION) HAVE BEEN ALLOCATED OR DERIVED. *).
   COMPLEMENTARY RELATIONSHIP:  TRACED ("FROM").
   SUBJECT ELEMENT_TYPE:  DECISION
                          ORIGINATING_REQUIREMENT.
   OBJECT ELEMENT_TYPE:  ALPHA
                         DATA
                         DECISION
                         ENTITY_CLASS
                         ENTITY_TYPE
                         EVENT
                         FILE
                         INPUT_INTERFACE
                         MESSAGE
                         OUTPUT_INTERFACE
                         PERFORMANCE_REQUIREMENT
                         R_NET
                         SUBNET
                         SUBSYSTEM
                         UNSTRUCTURED_REQUIREMENT
                         VALIDATION_PATH
                         VALIDATION_POINT
                         VERSION.
```

ATTRIBUTES

```
ATTRIBUTE:  ALTERNATIVES
            (* THE ALTERNATIVES THAT HAVE BEEN CONSIDERED TO
               RESOLVE A PROBLEM RESULTING IN A DECISION. *).
   APPLICABLE ELEMENT_TYPE:  DECISION.
   VALUE: TEXT.
```

```
ATTRIBUTE:  ARTIFICIALITY
                (* THE DEGREE OF FLEXIBILITY ALLOWED IN IMPLEMENTING
                   THE ELEMENT IN THE SOFTWARE. *).
   APPLICABLE ELEMENT_TYPE:  ALPHA
                             DATA
                             ENTITY_CLASS
                             ENTITY_TYPE
                             EVENT
                             FILE
                             INPUT_INTERFACE
                             MESSAGE
                             OUTPUT_INTERFACE
                             R_NET
                             SUBNET
                             VALIDATION_PATH
                             VALIDATION_POINT.
   VALUE: ARTIFICIAL
          (* THE ELEMENT HAS BEEN DEFINED FOR EXPLANATORY OR
             SIMULATION PURPOSES IN THE REQUIREMENTS STATEMENT
             AND NEED NOT BE PRESENT IN THE SOFTWARE. *).
   VALUE: VALIDATION
          (* THE ELEMENT IS NECESSARY FOR PERFORMANCE
             REQUIREMENTS EVALUATION BUT IS NOT REQUIRED IN THE
             OPERATIONAL SOFTWARE. *).
   VALUE: IMPLEMENT_PRECISELY
          (* THE ELEMENT MUST BE IMMPLEMENTED IN THE SOFTWARE
             EXACTLY AS DEFINED. *).
   VALUE: IMPLEMENT_APPROXIMATELY
          (* THE ELEMENT MUST BE IMPLEMENTED IN THE SOFTWARE,
             BUT THE PRECISE IMPLEMENTATION IS LEFT TO THE
             PROCESS DESIGNER. *).

ATTRIBUTE:  CHOICE
                (* THE ALTERNATIVE SELECTED TO SOLVE A PROBLEM
                   LEADING TO A DECISION.  THE RATIONALE FOR THE
                   CHOICE SHOULD BE INCLUDED HERE. *).
   APPLICABLE ELEMENT_TYPE:  DECISION.
   VALUE: TEXT.
```

```
ATTRIBUTE:  COMPLETENESS
            (* THE DEGREE TO WHICH THE DEFINITION OF AN ELEMENT IS
               IN FINAL FORM. *).
   APPLICABLE ELEMENT_TYPE:  ALPHA
                             DATA
                             DECISION
                             ENTITY_CLASS
                             ENTITY_TYPE
                             EVENT
                             FILE
                             INPUT_INTERFACE
                             MESSAGE
                             ORIGINATING_REQUIREMENT
                             OUTPUT_INTERFACE
                             PERFORMANCE_REQUIREMENT
                             R_NET
                             SOURCE
                             SUBNET
                             SUBSYSTEM
                             UNSTRUCTURED_REQUIREMENT
                             VALIDATION_PATH
                             VALIDATION_POINT
                             VERSION.
   VALUE: CHANGEABLE
          (* ALTHOUGH ALL RELATIONSHIPS, ATTRIBUTES, AND
             STRUCTURES MAY BE DEFINED FOR THE ELEMENT, SOME OF
             THEM WILL PROBABLY BE CHANGED.  INFORMATION ABOUT
             THE ELEMENT IS BELIEVED TO BE CORRECT, BUT IS
             SUBJECT TO CHANGE. *).
   VALUE: INCOMPLETE
          (* THE DEFINITION OF THE ELEMENT IS KNOWN TO BE
             INCOMPLETE.  THEREFORE, EVEN IF RELATIONSHIPS,
             ATTRIBUTES, AND STRUCTURES ARE STATED, THE ELEMENT
             DEFINITION IS STILL INCOMPLETE. *).
   VALUE: COMPLETE
          (* THE DEFINITION OF THE ELEMENT SHOULD BE ASSUMED TO
             BE COMPLETE AND WILL PROBABLY NOT CHANGE. *).
```

```
ATTRIBUTE:  DESCRIPTION
            (* ANY FREE FORM TEXTUAL MATERIAL DESCRIBING THE
               ELEMENT. *).
   APPLICABLE ELEMENT_TYPE:  ALPHA
                             DATA
                             DECISION
                             ENTITY_CLASS
                             ENTITY_TYPE
                             EVENT
                             FILE
                             INPUT_INTERFACE
                             MESSAGE
                             ORIGINATING_REQUIREMENT
                             OUTPUT_INTERFACE
                             PERFORMANCE_REQUIREMENT
                             R_NET
                             SOURCE
                             SUBNET
                             SUBSYSTEM
                             UNSTRUCTURED_REQUIREMENT
                             VALIDATION_PATH
                             VALIDATION_POINT
                             VERSION.
   VALUE: TEXT.
ATTRIBUTE:  ENTERED_BY
            (* THE IDENTITY OF THE LAST PERSON TO ENTER
               INFORMATION ABOUT THE ELEMENT. *).
   APPLICABLE ELEMENT_TYPE:  ALPHA
                             DATA
                             DECISION
                             ENTITY_CLASS
                             ENTITY_TYPE
                             EVENT
                             FILE
                             INPUT_INTERFACE
                             MESSAGE
                             ORIGINATING_REQUIREMENT
                             OUTPUT_INTERFACE
                             PERFORMANCE_REQUIREMENT
                             R_NET
                             SOURCE
                             SUBNET
                             SUBSYSTEM
                             UNSTRUCTURED_REQUIREMENT
                             VALIDATION_PATH
                             VALIDATION_POINT
                             VERSION.
   VALUE: TEXT.
ATTRIBUTE:  PROBLEM
            (* THE PROBLEM THAT HAS LED TO THE NEED FOR A
               DECISION. *).
   APPLICABLE ELEMENT_TYPE:  DECISION.
   VALUE: TEXT.
```

4.0 CONTROLLING REVS

REVS is a unified collection of software composed of functions that contribute in different ways to the development of requirements. (The capabilities of REVS and the functions which provide them are summarized in Section 2.0.) The user controls REVS using the REVS Control Language (RCL). In general, the user input to REVS consists of a continuous stream of RCL commands which activate the various functions and direct them to perform certain operations. The functions have individual RCL commands for their control; in the case of the RSL translation and RSL extension (RSLXTND) functions these commands consist of RSL. Thus the input stream to REVS can be depicted simply as the following:

```
Activation of function 1

     -)
     -} Commands to function 1
     -)

Activation of function 2

     -)
     -} Commands to function 2
     -)

  •
  •
  •

Activation of function n

     -)
     -} Commands to function n
     -)
```

where functions 1 through n refer to any of the REVS functions.

Activation of a REVS function is controlled by the REVS Executive. It is always active from initiation to termination of REVS execution regardless of the individual REVS functions which are activated throughout an execution. The activation of a REVS function is requested by user command to the Executive. Other commands to the REVS Executive are available to control the general operation of REVS. Most of these may appear anywhere in the input stream including within the input to individual functions where they will be intercepted and executed by the Executive. Thus the user input commands to REVS can be categorized into the following types:

- Executive Control Statements which activate REVS functions and direct the operational characteristics of REVS.
- Function Control Statements which are received by the Executive and routed to the activated function which processes each statement and performs the desired operations.

This organization, as illustrated in Figure 4-1, allows complete flexibility in the type of operations that can be performed in a single execution of REVS. The Executive Control Statements are described in this section and permit the user to activate the REVS functions, to control the input of information to REVS, cause mode changes in the operation of REVS, control the content and disposition of REVS output, and terminate the execution of REVS. The Function Control Statements and their use are described in subsequent sections of this document (Sections 5 through 8).

REVS Operating Modes

REVS has two modes of operation, termed off-line and on-line. In the off-line mode, the user inputs are made through the system input stream which usually originates on cards. In the on-line mode, communication between the user and REVS is interactive through an ANAGRAPH keyboard-display console.

Executive Control Input Format

The input format of the Executive Control Statements is the same in the off-line or on-line modes and requires that each statement be solely contained on one line, that only one statement per line occur, and that no other information precede the statement on the line. Any information following the statement on the same line is treated as commentary. The syntax of each of the REVS Executive Control Statements is presented in this section along with a description of the resulting action by REVS. The syntax is presented in extended Backus-Naur Form (BNF). Readers unfamiliar with BNF should read Appendix A before proceeding (the syntax of all RCL and RSL is presented throughout the text in this notation). **Appendix C contains the complete syntax for the Executive Control Statements, in both BNF and in the form of syntax diagrams.**

Executive Messages

The REVS Executive will output messages to the user containing information about actions of the Executive. These messages are documented

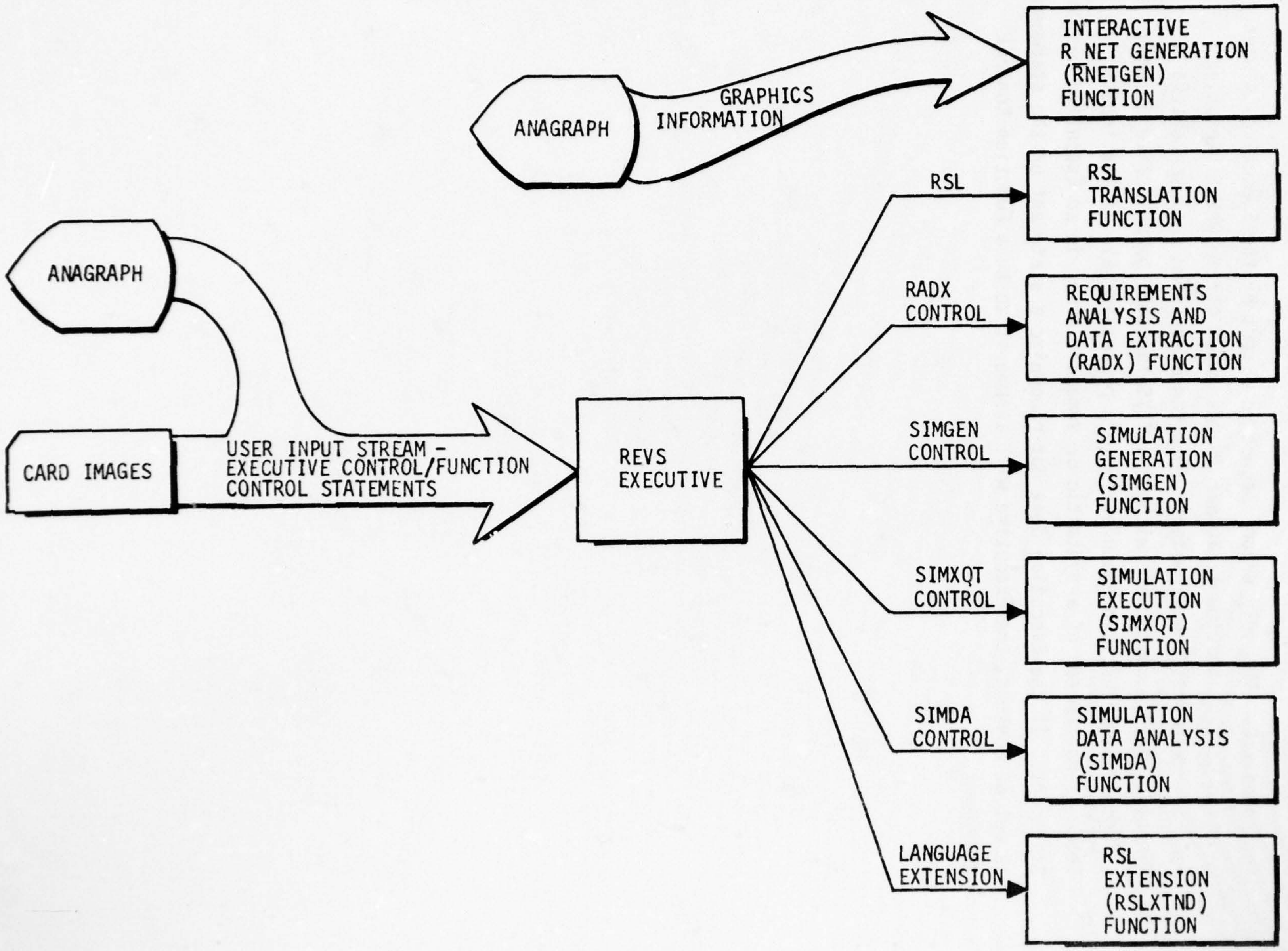

ANAGRAPH
GRAPHICS INFORMATION
INTERACTIVE R NET GENERATION (RNETGEN) FUNCTION
ANAGRAPH
CARD IMAGES
USER INPUT STREAM - EXECUTIVE CONTROL/FUNCTION CONTROL STATEMENTS
REVS EXECUTIVE
RSL
RSL TRANSLATION FUNCTION
RADX CONTROL
REQUIREMENTS ANALYSIS AND DATA EXTRACTION (RADX) FUNCTION
SIMGEN CONTROL
SIMULATION GENERATION (SIMGEN) FUNCTION
SIMXQT CONTROL
SIMULATION EXECUTION (SIMXQT) FUNCTION
SIMDA CONTROL
SIMULATION DATA ANALYSIS (SIMDA) FUNCTION
LANGUAGE EXTENSION
RSL EXTENSION (RSLXTND) FUNCTION

Figure 4-1 Information Flow of User Inputs

in Appendix C. The only error diagnostic output by the Executive is when Function Control Statements appear in the input stream where Executive Control Statements are expected. This can occur when the user enters Function Control Statements without a REVS function being active or when a function terminates prematurely before processing all of its inputs. There is no concept of a syntactic or semantic error in an Executive Control Statement. If the Executive does not recognize a statement or the statement is out of context, the Executive will assume it to be a Function Control Statement.

4.1 SELECTING A REVS FUNCTION

At the start of a REVS execution no function is active. The FUNCTION statement is used to specify to the REVS Executive which function to activate. The syntax of the statement is:

```
[FUNCTION] function-name.
```

In the statement, function-name is defined as one of the following:

Function Name	REVS Function
RSL	RSL Translation (requirements translation)
RNETGEN	Interactive R-Net Generation
RADX	Requirements Analysis and Data Extraction
SIMGEN	Simulation Generation
SIMXQT	Simulation Execution
SIMDA	Simulation Data Analysis
RSLXTND	RSL Extension Translation

The FEND statement is used to explicitly terminate an activated REVS function. The syntax of the statement is:

```
FEND.
```

An active REVS function may also be terminated implicitly by entry of another FUNCTION statement, by change of the operating mode (a GO statement changes the mode, see Section 4.3), and by termination of REVS execution (see STOP statement, Section 4.5).

Implicit termination applies to all of the REVS functions except RNETGEN. The RNETGEN function has special features that are applicable only in the on-line operating mode and special control is required to terminate this function. This is described in Section 5.2.

4.2 CONTROLLING INPUT

The primary source of input for REVS is determined by the operating mode. When operating on-line, the user communicates interactively with REVS by way of the ANAGRAPH console keyboard and trackball. In the off-line mode, the user inputs are made through the system input stream. In either mode, the user may control the filtering of the input for Executive Control Statements and may direct the Executive to temporarily use an alternate source of input.

4.2.1 Avoiding Executive/Function Statement Conflicts

Occasionally, the content of a Function Control Statement might have the appearance of an Executive Control Statement. (For example, a line image in an RSL text string might begin with keywords and punctuation which form a legal Executive command.)

The user may direct the Executive to enter a transparent data mode, thus avoiding misinterpretation of input, for the current input medium by using the TRANSPARENT statement. This statement has the form:

TRANSPARENT string.

where the string contains from one to eight characters. These may be any characters except blank and period.

Upon entry of a TRANSPARENT statement, no further examination for Executive Control Statements in subsequent input is made until an image is encountered which starts with the specified string. The terminating image is logged, but is not treated as input by REVS.

If the TRANSPARENT statement is encountered in an alternate input source (an ADDFILE, see below) the transparency will be observed until the transparency string is encountered or until the end of file on the ADDFILE is reached.

4.2.2 Designating an Alternate Input Source

The ADDFILE statement is used to cause REVS to accept input from a specified file. The syntax of the statement is:

ADDFILE [TRANSPARENT] access-name.

The access name is the name of a file and is limited to eight alphanumeric characters on the ASC. The specified file is read in its entirety as though it were inserted in the primary input stream in place of the ADDFILE statement.

All Executive commands are valid within the file, except another ADDFILE statement. Should a GO command (see Section 4.3) be placed in the file, the GO operation is performed and the ADDFILE command stays in effect until all inputs are read from the file. The TRANSPARENT option to the ADDFILE statement causes suppression of the examination of the file for Executive Control Statements.

4.3 SELECTING REVS OPERATING MODE

As stated previously, REVS is either operated off-line or on-line via the ANAGRAPH color console. When REVS is initially executed, the operating mode is off-line. The GO command, which has the following syntax, is used to change the operating mode.

```
   ┌ONLINE  ┐
GO │OFFLINE │ [ONLY]. [display-remark]
   └OPPOSITE┘
```

An example of this statement is:

```
GO ONLINE. JOE USER -- PHONE 837-2400.
```

The keywords within the GO command determine the new operating mode. ONLINE means that the ANAGRAPH console will be enabled and the next input to REVS will be from there. OFFLINE means that the ANAGRAPH will be disabled and the next input to REVS will be from the system input stream. (Note: If a GO statement is entered requesting the same mode as the current mode, this statement will be treated as a Function Control Statement by the Executive.) The OPPOSITE option causes the operating mode to change from on-line to off-line, or from off-line to on-line, depending on the current mode. The OPPOSITE option is assumed if no option is specified. The ONLY option causes an irrevocable change to the specified mode. If another GO statement is entered subsequent to the ONLY option, it will terminate the execution of REVS (i.e., it will be interpreted as a STOP statement, see Section 4.5). **See Section 10 for the installation dependencies of this feature.**

When entering the on-line mode, the display remark portion of the GO statement is output as an on-line identification. It is displayed on the ANAGRAPH with the TRW logo when a console is ready for use and should contain sufficient information to identify and locate the user requesting the console.

When operating in the on-line mode, the user communicates interactively with REVS by way of the ANAGRAPH console keyboard and trackball. The keyboard consists of a panel containing a typical set of alphanumeric characters which the user may key in when the keyboard has been enabled for input. A short red horizontal cursor will be displayed when the keyboard has been enabled. The keyboard ENTER key is depressed to terminate each line of input and transfer it to REVS. The trackball consists of a movable white cursor operated by a hand-rotated ball and an entry key at the right of the console. This facility provides the user the capability to select (input) any coordinate position on the face of the CRT. The trackball is available for input when the white cursor is blinking. By rotating the trackball, thus moving the white blinking cursor to any desired position on the CRT, and subsequently depressing the trackball entry key, the user inputs the selected point to REVS.

When the logo appears on the screen with the user's on-line identification, the enter key next to the trackball is depressed to start operating REVS. A display will then appear on the screen as depicted in Figure 4-2. The lowest line of the screen is the REVS input area, the line above it is the echo of the last line entered and the third line is the REVS status display. The uppermost part of the screen (with heavy green border) is the area where output lines are displayed for viewing. As REVS generates output, the page display will wrap over itself from top to bottom with a red line underlining the last line displayed. This permits continual visibility of portions of the previous output (except when interrupted by use of the RNETGEN function).

The status line, annotated in blue by REVS, has five display fields to show the current status of REVS. The first variable is the name of the currently activated function or REVSEXEC if no function is activated. The second variable displayed is the current REVS input file name. This will be $ONLINE$ to indicate the on-line keyboard, except when input has been diverted through an ADDFILE statement, in which case it will be the access name of the file being read. The third variable displayed is the output routing status which is set by the OUTPUT statement (see Section 4.4) and indicates whether output lines are being displayed on-line and/or off-line. The fourth variable is the logging resolution status, and indicates ALL or EXECRCL according to the last LOG statement (see Section 4.4). The final field is the transparency indicator which displays the required transparency terminator string when in the transparent mode.

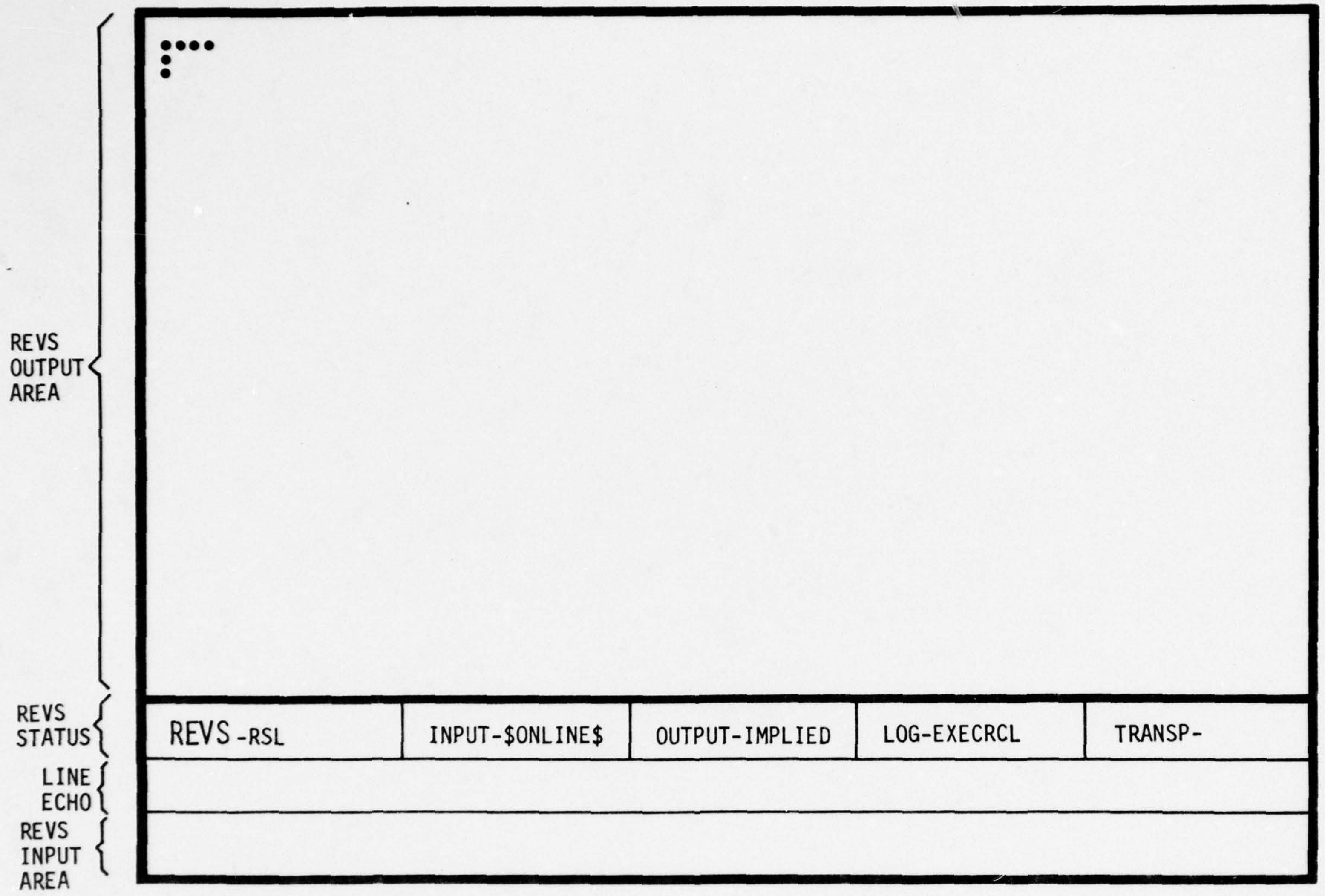


Figure 4-2 Nominal REVS Display

4.4 CONTROLLING OUTPUT

There are two primary output files generated by REVS. The first is an Execution Log (REVS.LOG) of the activities that took place during the execution of REVS. The second file contains the output generated by each function that was executed and is identified as REVS.OUT.

4.4.1 Controlling the Logging Resolution

The type of information that is placed in REVS.LOG is controlled by the LOG statement which has the following syntax:

$$\text{LOG}\begin{bmatrix}\text{ALL}\\ \text{EXECRCL}\end{bmatrix}.$$

The normal information output to REVS.LOG is Executive Control Statements. This information is identical to that provided when the EXECRCL option is selected or when a LOG statement with no option is input. The ALL option causes both Executive Control Statements and Function Control Statements to be logged.

An example of REVS.LOG is presented in Figure 4-3. Each line is coded F, X, or M to respectively denote a Function Control Statement, an Executive Control Statement, or a message issued by REVS. Also identified for each control statement is the input source, either $ONLINE$ for the ANAGRAPH, REVSIN for the system input stream, or the access name for an ADDFILE.

4.4.2 Controlling the Routing of Function Output

The OUTPUT statement is used to control the routing of primary output (file REVS.OUT) from the REVS functions. The syntax of the statement is:

$$\text{OUTPUT}\begin{bmatrix}\text{ONLINE}\\ \text{OFFLINE}\\ \text{BOTH}\\ \text{IMPLIED}\end{bmatrix}.$$

The initial routing is IMPLIED, i.e., output is sent to the off-line printer when operating in the off-line mode and to the interactive console when in the on-line mode. The OFFLINE option directs REVS to only use the off-line printer regardless of the operating mode. The ONLINE option directs REVS to only use the interactive console for subsequent output. If

```
    06/27/77
M   09:36:00             XX 000  REVS   BASELINE VERSIØN = 11, (DATE=06/22/77,  TIME=17:08:41)
 X  09:36:02   REVSIN            RSL.
M   09:36:03             XX 001  FUNCTIØN RSL       INITIATED.
 X  09:42:36   REVSIN            FEND. (GENERATED STATEMENT)
M   09:42:37             XX 002  FUNCTIØN RSL       CØMPLETED.
 X  09:42:37   REVSIN            SIMGEN.
M   09:42:37             XX 001  FUNCTIØN SIMGEN    INITIATED.
 X  09:42:41   REVSIN            FEND. (GENERATED STATEMENT)
M   09:48:02             XX 002  FUNCTIØN SIMGEN    CØMPLETED.
 X  09:48:03   REVSIN            SIMXQT.
M   09:48:03             XX 001  FUNCTIØN SIMXQT    INITIATED.
 X  09:48:04   REVSIN            FEND. (GENERATED STATEMENT)
M   09:48:04             XX 002  FUNCTIØN SIMXQT    CØMPLETED.
 X  09:48:04   REVSIN            SIMDA.
M   09:48:04             XX 001  FUNCTIØN SIMDA     INITIATED.
 X  09:48:05   REVSIN            FEND. (GENERATED STATEMENT)
M   09:48:06             XX 002  FUNCTIØN SIMDA     CØMPLETED.
 X  09:48:06   REVSIN            STØP.
M   09:48:09             XX 007  REVS CØMPLETED: NØRMAL TERMINATIØN.
```

Figure 4-3 REVS.LOG Example Listing

the OUTPUT ONLINE option is selected when operating off-line, the user will not receive any output from REVS. The BOTH option directs subsequent output to be routed by REVS to both the interactive console (when in on-line mode) and the off-line printer. An example of a REVS.OUT listing as it appears on the printer is shown in Figure 4-4.

4.4.3 Controlling Pagination

The user has the option to perform page control of the REVS.OUT file with the NEWPAGE statement.

$$\text{NEWPAGE} \begin{bmatrix} \text{OFFLINE} \\ \text{ONLINE} \\ \text{BOTH} \\ \text{IMPLIED} \end{bmatrix} . \text{ [offline-page-titling-remark]}$$

An example of this statement is:

NEWPAGE. START INPUTS FOR ENTITY CHANGES.

The selection of the OFFLINE option causes the printer to skip to the top of the next page and to display the offline-title (which is limited to 60 characters). The ONLINE option causes the screen to be cleared and the next output line to appear at the top of the screen. The offline-title has no effect on the on-line display. The BOTH option makes both of the above actions occur. When no option is specified or when the IMPLIED option is specified, the action that is taken is determined by the current output routing.

4.4.4 Displaying Information On-Line

When a page display is completed during on-line operation, the user is presented the following response prompting message at the bottom of the screen:

CONTINUE, INTERRUPT, OUTPUT OFFLINE, NONSTOP.

One of the options contained in this statement is selected via a trackball entry to control subsequent actions on the output displayed by REVS. The CONTINUE option causes the next page of output to be displayed. The INTERRUPT entry causes a long display to be truncated if the RADX function is activated. If input is being read from an ADDFILE, an INTERRUPT entry will terminate the reading of the ADDFILE. An OUTPUT OFFLINE selection causes the current display and subsequent displays to only be routed

```
XX 000  REVS   BASELINE VERSION = 10+, (DATE=05/13/77,  TIME=10:41:22)
        RSL.

XX 001  FUNCTION RSL       INITIATED.  ************************************
MODIFY PERFORMANCE_REQUIREMENT RADIATED_POWER.
REMOVE TEST.
INSERT
        TEST:
        "CONST
         RADIATED_POWER_LIMIT=5.0; (* TEMPORARY REPLACEMENT FOR (TBD) *)
         VAR
         RADAR_POWER: REAL;
         INTERVAL: REAL;
         BEGIN
          RADIATED_POWER:=TRUE;
          RETRIEVE FIRST RECORDING FOR STARTING_POINT;
          FOR EACH RADAR_COMMAND_OUTPUT_POINT RECORDING
          DO
            RADAR_POWER:=0.0;
            INTERVAL:=RADAR_COMMAND_OUTPUT_POINT.TRANSMIT_START;
            FOR EACH RADAR_COMMAND_OUTPUT_POINT RECORDING
              SUCH THAT ((RADAR_COMMAND_OUTPUT_POINT.TRANSMIT_START<=
                          INTERVAL+1.0) AND
                         (RADAR_COMMAND_OUTPUT_POINT.TRANSMIT_START>=
                          INTERVAL))
            DO
              SELECT FIRST RECORD FROM STARTING_POINT.WAVEFORM_TABLE
                SUCH THAT (RADAR_COMMAND_OUTPUT_POINT.RADAR_TYPE=
                           STARTING_POINT.WF_NAME));
                IF RECORD_FOUND THEN
                   RADAR_POWER:=RADAR_POWER+
                                STARTING_POINT.WF_CHARACTERISTICS;
            ENDFOREACH;
            IF (RADAR_POWER>RADIATED_POWER_LIMIT) THEN
              RADIATED_POWER:=FALSE;
          ENDFOREACH
         END;".
MODIFY SUBSYSTEM SSPERMRL.
REMOVE TRACED FROM:
  TLS_DPSPR_SUBSECTION_3_2_5_FUNCTIONAL_REQUIREMENTS.
DELETE SUBSYSTEM SSPERMRL.
MODIFY SUBSYSTEM SSPERMFL.
INSERT TRACED FROM:
  TLS_DPSPR_SUBSECTION_3_2_5_FUNCTIONAL_REQUIREMENTS.
MODIFY DATA D1_WINDOW_DATA.
INSERT INPUT TO ALPHA SET_LOST.
MODIFY DATA D1_XMIT.
INSERT INPUT TO ALPHA SET_LOST.
MODIFY DATA XMIT_START.
INSERT INPUT TO ALPHA PICK_COMMAND.
MODIFY DATA D1RTN_ERROR_REPORT.
INSERT USE BOTH.
MODIFY DATA RANGE_MARK_INFORMATION.
INSERT USE BETA.
MODIFY DATA REASON_FOR_TRANSMISSION_FAILURE.
INSERT USE BOTH.
MODIFY DATA RECEIVE_INFORMATION.
INSERT USE BETA.
XX 002  FUNCTION RSL       COMPLETED.  ************************************
        STOP.

XX 007  REVS COMPLETED: NORMAL TERMINATION.
```

Figure 4-4 REVS.OUT Example Listing

off-line until changed by an OUTPUT control statement. The NONSTOP option causes continuous display of REVS output lines, without a pause after each page is displayed. This process continues until the user is prompted for more input or until the enter key on the trackball is pressed, in which case the NONSTOP mode is discontinued.

4.5 TERMINATING EXECUTION OF REVS

The STOP statement is used to terminate the execution of REVS. The statement has the following syntax:

$$\text{STOP}\begin{bmatrix}\text{JOB}\\ \text{STEP}\end{bmatrix}\ .\ \text{[display remark]}$$

This statement can be entered either in the on-line or off-line mode. When entered on-line, the display remark is written as a terminating message on the final REVS ANAGRAPH display. If neither the JOB or STEP option is specified, the STEP option is assumed. Also, an actual end of file on the REVS primary input stream implies STOP STEP. The STEP option causes termination of REVS and its job step, but allows normal execution of any remaining job steps. The JOB option causes the REVS step and all subsequent steps to be cancelled.

5.0 BUILDING A REQUIREMENTS DATA BASE

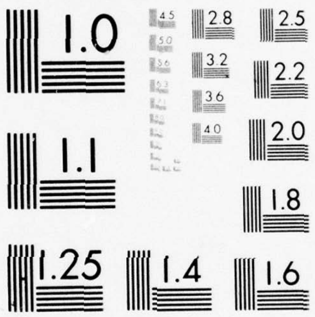
REVS provides two functions for establishing and maintaining a requirements data base. The primary one, the RSL translation function, accepts RSL statements and enters, modifies, or deletes requirements data base information as specified by these statements. The RSL syntax for defining, modifying, and deleting entries in the data base and examples illustrating the use of RSL are presented in Section 5.1.

The Interactive R-Net Generation (RNETGEN) function augments the capabilities of the RSL translation function by allowing the user to enter and modify data base structures in a two-dimensional graphic form. The RNETGEN function also provides the capability to establish, either automatically or under user control, graphics coordinate data for structures entered in the form of RSL. The operation and use of RNETGEN is described in Section 5.2.

The basic unit of requirements as stated in RSL is the element. The user designates elements by unique names and establishes their meaning in the requirements by stating their attributes, relationships and structures. REVS imposes certain restrictions on the selection of names for the requirements elements. The user should not use the keywords appearing in the RSL and RCL syntax; in addition, if the requirements are to be simulated, the user should not use as element names the keywords of PASCAL. The keywords for RSL, RCL, and PASCAL are listed in Appendix B. When simulating requirements, REVS also requires that certain element names be unique in a limited number of characters. These restrictions are installation dependent and are detailed in Section 10.

AD-A046 572 TRW DEFENSE AND SPACE SYSTEMS GROUP HUNTSVILLE ALA F/G 9/2
REVS USERS MANUAL. SREP FINAL REPORT. VOLUME II.(U)
AUG 77 M E DYER, L J GUNTHER, R W SMITH DASG60-75-C-0022
UNCLASSIFIED TRW-27332-6921-026-VOL-2 NL

2 OF 5
AD
A046572


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

5.1 ENTERING REQUIREMENTS IN RSL (RSL FUNCTION)

This section describes the use of the RSL function to enter requirements stated in the Requirements Statement Language (RSL). There are three types of statements, called RSL commands, which the user can input to define new elements, modify existing elements, and delete elements. Two additional commands allow for changing the name and the element type of an element. These five commands are presented and illustrated in the following subsections. The command syntax presented is expressed in the extended Backus-Naur Form (BNF) explained in Appendix A. The complete RSL syntax from which these rules were extracted is presented in Appendix D, in both BNF and syntax diagram forms.

General information that applies to all commands input to the RSL function and an explanation of the output from the RSL function are provided below.

<u>RSL Input Specifications</u>

The following rules summarize the input format accepted by the RSL function. These rules, some of which are given in greater detail in Appendix B, together with the syntax summarized in Appendix C, give a complete input specification for the RSL function.

- Each RSL statement is terminated by a period that is not contained within a comment or text string.
- Only the first 72 characters of each input line are significant; all other characters are ignored.
- All names have a maximum length of 60 characters. The first character in a word must be a letter or an underscore; remaining characters must be letters, underscores, or digits.
- All numbers are in standard PASCAL form (see Appendix B).
- All names and numbers are terminated by one or more blanks, or punctuation marks (an end of input record is equivalent to a blank).
- A comment, a sequence of characters beginning with (* and ending with *), may only be used where specified in the RSL syntax.
- A text string, a sequence of characters beginning and ending with double quotes, may only be used where specified in the RSL syntax.

- The comma, colon, and semicolon are optional punctuation marks that are equivalent to a blank.

- Relation optional words may appear anywhere in the input and will be ignored.

RSL Output Specifications

For each input line, the RSL function will output the following on REVS.OUT:

- An echo of the 72 significant characters, a vertical bar ("|"), and the remaining characters of the input line (the characters to the right of the vertical bar are ignored by the RSL function).

- If an error was detected in the input line, an additional line will be output. For each error detected an up-arrow ("↑"), followed by an error number will be output. These error numbers and their associated meanings are listed in Section 5 of Appendix D. If two or more errors are detected at a symbol, only one up-arrow will be output and the error numbers will be separated by commas.

In addition to the standard output described above, if any errors were detected by the RSL function, the following message will be produced on the REVS.LOG file:

```
TT  001  NUMBER OF TRANSLATION ERRORS = XX
```

where XX is the number of errors detected.

There are a few cases in which the RSL function may not be able to recover from an error. This will normally occur only if there are machine errors or severe errors in the RSL input, or if the ASSM or required input files DONNEES and RSLDICT are missing or unusable. If any of these occur, the RSL function will output the words "FATAL ERROR" followed by one of the fatal error diagnostics given for error numbers 600 through 606 in Section 5 of Appendix D. The following message will also be placed in the REVS.LOG file:

```
TT  002  FATAL ERROR IN TRANSLATION.
```

5.1.1 Defining a New Element

The explicit definition of a new element consists of a declaration of the element, optionally followed by a series of element definition sentences which declare attribute values, relationships, and a structure or path for the element. The syntax for a new element definition is:

[DEFINE] element-type-name element-name [comment].

$\left\{\text{[INSERT] <element definition sentence>}\right\}_0^n$

As shown in the syntax, the word DEFINE is optional in a new element definition. Its use is, however, recommended. If DEFINE is used and the specified element name is currently defined in the ASSM an error will be reported. If DEFINE is not used the existing element will be modified without notification to the user. The word INSERT is also optional preceding each element definition sentence and may be used to improve the readability of the input. The omission of the word INSERT has no affect on the interpretation of the input.

There are four types of element definition sentences: the attribute, relation, path, and structure declarations. These declarations are discussed in subsections below following the discussion of the new element declaration. Another type of new element definition, termed an implicit definition, occurs whenever a previously undefined element is introduced within a relation, path, or structure declaration. This implicit declaration is discussed separately in Section 5.1.1.6.

The top-level syntax for an explicit new element definition is:

<new element definition>::=

[DEFINE] element-type-name element-name [comment].

$\left\{\text{[INSERT] <element definition sentence>}\right\}_0^n$

```
<element definition sentence>::=
    <attribute declaration>
  | <relation declaration>
  | <path declaration>
  | <structure declaration>
```

5.1.1.1 Declaring a New Element

A new element is declared by optionally specifying the word DEFINE, followed by an element type name for the element, an element name, and an optional comment for the element.

[DEFINE] element-type-name element-name [comment].

For example, the following are all valid declarations of new elements:

```
DEFINE ALPHA PROCESS_VALID_RETURNS.
DEFINE DATA OBJECT_ID (*UNIQUE IDENTIFIER*).
DEFINE R_NET ALLOCATE_RESOURCES.
```

The first declaration above defines an element named PROCESS_VALID_RETURNS of type ALPHA. The second declaration defines the element OBJECT_ID of type DATA. The third declaration defines the element ALLOCATE_RESOURCES of type R_NET. Elements PROCESS_VALID_RETURNS and ALLOCATE_RESOURCES are defined with no associated comments; OBJECT_ID is defined with a comment.

The syntax for declaring a new element is the first part of the complete syntax for a new element definition.

```
<new element definition>::=
    [DEFINE] element-type-name element-name [comment].
```

5.1.1.2 Declaring an Attribute Value

As noted above, one of the types of sentences which may appear within a new element definition is an attribute declaration, optionally preceded by the word INSERT. An attribute declaration declares an attribute value for the element by giving the name of the attribute followed by the desired attribute value and an optional comment.

$$[\text{INSERT}]\ \text{attribute-name} \left\{ \begin{array}{l} \text{value-name} \\ \text{number} \\ \text{text-string} \end{array} \right\}_1^1 [\text{comment}].$$

The following example declares several attribute values for the element OBJECT_ID.

```
DATA OBJECT_ID.
    TYPE INTEGER.
    INITIAL_VALUE 0 (*PRESET VALUE*).
    USE BOTH.
    DESCRIPTION "IDENTIFIER FOR AN OBJECT".
```

Any number, including zero, of attribute declarations may be given for an element; each one may be preceded by the word INSERT.

Note that once an element has a value for a particular attribute, it is always necessary to remove that value before a new value for the same attribute can be declared. The process of modifying element definitions is discussed in Section 5.1.2.3.

Forms of Attribute Values

An RSL attribute is defined in terms of an attribute name, a set of one or more element types to which the attribute may apply, and a set of allowable values. These allowable values are either particular value names or one of the value class names NAMED, NUMERIC, or TEXT. The value class names have the following interpretations:

NAMED	- The value in an attribute declaration may be any name not used in a different context.
NUMERIC	- The value in an attribute declaration may be any signed or unsigned integer or real number (see Appendix B).
TEXT	- The value in an attribute declaration may be any text string (a sequence of characters enclosed in double quotes).

For example, attribute LOCALITY has legal values GLOBAL and LOCAL, meaning that the only values which may be specified for LOCALITY are the names GLOBAL or LOCAL. The attribute INITIAL_VALUE has legal values NAMED and NUMERIC indicating that any name or number may be specified.

The syntax for declaring an attribute value is:

[INSERT] <attribute declaration>

<attribute declaration>::=

$$\text{attribute-name} \left\{ \begin{array}{l} \text{value-name} \\ \text{number} \\ \text{text-string} \end{array} \right\}_1^1 \text{[comment].}$$

5.1.1.3 Declaring a Relationship Instance

A relationship is established between the subject element of the new element definition and some other object element by specifying a relation declaration optionally preceded by the word INSERT. The relation declaration gives the relationship name, perhaps followed by the appropriate relation optional word, followed by the types and names of one or more object elements and optional comments.

[INSERT] relation-name [relation-optional-word]

$$\left\{ \text{[element-type-name] element-name [comment]} \right\}_1^n.$$

The following example declares several relationships between DATA STATE and other elements.

```
DATA STATE.
    INPUT TO ALPHA UPDATE_STATE.
    INCLUDES DATA X
             DATA Y (*NORTH*)
             DATA Z (*UP*).
    CONTAINED IN FILE STATE_HISTORY (*RECORD OF STATE CHANGES*).
```

Each comment specified in a relationship declaration is associated with exactly one relationship instance. Thus the INCLUDES declaration shown above establishes three instances of the INCLUDES relationship, only two of which have comments; i.e., relationship instances between DATA STATE and each of the DATA elements X, Y, and Z are established but only the relationship instances to Y and Z have associated comments.

On the assumption that the element named in a relationship declaration has already been defined, the element type name need not be specified. Thus,

```
INPUT TO UPDATE_STATE.
```

is equivalent to

```
INPUT TO ALPHA UPDATE_STATE.
```

if an ALPHA with the name UPDATE_STATE has been previously defined.

As in other declarations about an element, the optional word INSERT may be specified preceding the relationship name to improve the readability of the input.

The appearance of an element type name and an element name in a relationship declaration may serve as an implicit declaration of the element name. The reader is referred to Section 5.1.1.6 for a discussion of this type of declaration.

<u>Use of Relationship and Complementary Relationship Names</u>

A relationship in RSL is treated like a non-commutative binary relation; that is, a statement of an association between a subject element and an object element which are distinct. For each relationship there is a defined set of element types which may serve as subjects and a defined set of element types which may serve as objects. For each relationship there is also defined a complementary relationship which is the converse of the relationship in the sense that the subject set of the relationship is the object set of the complementary relationship and the object set of the relationship is the subject set of the complementary relationship.

This duality of relationships-complementary relationships is important in the statement of relationship declarations since the declaration itself specifies only the object element; the subject element is taken to be the element declared in the preceding element declaration. If this subject element is in the subject set for a relationship, then the relationship name should be used. Conversely, if this subject element is in the object set for a relationship, then the complementary name should be used. For example:

```
ALPHA UPDATE_STATE.
    INPUTS DATA STATE.
```

establishes the same relationship instance as:

```
DATA STATE.
    INPUT TO ALPHA UPDATE_STATE.
```

The syntax for declaring a relationship instance is:

[INSERT <relation declaration>

<relation declaration>::=
relation-name [relation-optional-word]
$\left\{\text{[element-type-name] element-name [comment]}\right\}_1^n$.

5.1.1.4 Declaring a Net Structure

A net structure may be declared for any RSL element which is of type R_NET or SUBNET by specifying a structure declaration, optionally preceded by the word INSERT. The structure declaration itself consists of the word STRUCTURE followed by two or more node constructs, the word END and an optional comment:

[INSERT] STRUCTURE $\left\{\text{<node>}\right\}_2^n$ END [comment].

There are two classes of node constructs, primitive and complex. The primitive nodes are single entry and single exit constructs. The complex nodes are multiple entry or multiple exit constructs and express information about the sequencing of the nodes they contain.

The primitive nodes are of three types: the element node, the terminator node, and the SELECT node. The complex nodes are of four types: the FOR EACH node, the AND node, and two types of OR nodes. Each of these node types is discussed below.

Formally, a node is defined as:

```
<node>::=
            <element node>
         |  <terminator>
         |  <select node>
         |  <for-each node>
         |  <and node>
         |  <or node>
         |  <consider-or node>
```

Rules Regarding Structures

The following general rules are enforced for structure declarations. Additional rules relevant to particular node types are specified in the discussions of the individual node types.

1. Each path through a structure must be terminated by a terminator node or by an OUTPUT_INTERFACE element node.
2. The only place an INPUT_INTERFACE element node may appear is as the first node on a structure for an R_NET.
3. A RETURN terminator node may only appear on a structure for a SUBNET and exactly one RETURN must appear on each SUBNET structure.
4. A structure may only be declared for an R_NET or a SUBNET.
5. No more than one structure may be declared for any R_NET or SUBNET.

Element Node

An element node consists of an element type name followed by an element name and an optional comment. If the element has been previously defined, the element type name may be omitted. If the element has not been defined, the element type name must be specified and the element is implicitly defined as discussed in Section 5.1.1.6. In either case, the comment is associated with the structure node, not with the element itself. Element nodes may only be constructed from elements of a type which is defined with STRUCTURE APPLICABILITY NET. These types are ALPHA, EVENT, INPUT_INTERFACE, OUTPUT_INTERFACE, SUBNET, and VALIDATION_POINT.

A simple two-node structure using only element nodes is illustrated below.

```
R_NET ACCEPT_RETURN.
    STRUCTURE
        ALPHA CHECKER (*CHECK VALIDITY OF DATA*)
        OUTPUT_INTERFACE POST_COMMAND
    END.
```

Formally, an element node is defined as:

```
<element node>::=
    [element-type-name] element-name [comment]
```

Rules for Element Nodes

The following rules regarding element nodes are enforced.

1. An INPUT_INTERFACE element node may only appear as the first node on an R_NET structure.
2. An INPUT_INTERFACE element node may not appear on a SUBNET structure.
3. An OUTPUT_INTERFACE element node terminates a path of a structure.

Terminator Node

A terminator node or OUTPUT_INTERFACE is used as the final node on a path of a structure. Two types of terminator nodes are defined, the RETURN node used on a structure for a SUBNET to indicate the point at which the SUBNET returns to its calling structure, and the TERMINATE node used to indicate the termination of a structure branch. The syntax of a terminator node is the word TERMINATE or RETURN, optionally followed by a comment for the node:

```
    TERMINATE [comment]
|   RETURN [comment]
```

Two simple examples of structures using only element nodes and terminator nodes are given below.

```
SUBNET SUB1.
    STRUCTURE
        ALPHA CHECKER
        SUBNET DOER
        RETURN
    END (*CHECK DATA AND PERFORM ACTIONS*).
R_NET NET1.
    STRUCTURE
        SUBNET SUB1
        TERMINATE (*END OF NET1*)
    END.
```

Formally, a terminator node is defined as:

```
<terminator>::=
        TERMINATE [comment]
    |   RETURN [comment]
```

Rules for Terminator Nodes

The following rules are enforced for terminator nodes.

1. Exactly one RETURN node must appear on each SUBNET structure.

2. A RETURN node may not appear on an R_NET structure.

3. A TERMINATE or RETURN node terminates a path on a structure.

SELECT Node

A SELECT node is a primitive node which selects an entity based upon the value of a Boolean expression. The syntax for a select node is:

$$\text{SELECT} \left\{ \begin{array}{l} \text{[ENTITY\_CLASS] entity-class-name} \\ \text{[ENTITY\_TYPE] entity-type-name} \end{array} \right\}_1^1$$

```
    SUCH THAT <condition> [comment]
```

Thus, the basic form for a select node is one of the following:

```
SELECT ENTITY_CLASS entity-class-name
    SUCH THAT (<Boolean expression>)
```

or

```
SELECT ENTITY_TYPE entity-type-name
    SUCH THAT (<Boolean expression>)
```

In addition, a comment may be placed after the (<Boolean expression>). Also, if the entity class name or entity type name is already defined, the preceding element type name may be omitted. For example, the following are valid SELECT nodes.

```
SELECT ENTITY_CLASS LARGE SUCH THAT (X_ERROR >= ERROR_LIMIT)

SELECT ENTITY_TYPE SMALL SUCH THAT (X*Y < LIMIT) (*EXAMPLE*)
```

Assuming that ENTITY_CLASS LARGE and ENTITY_CLASS SMALL are defined, the following two SELECT nodes are equivalent to those above.

```
SELECT LARGE SUCH THAT (X_ERROR >= ERROR_LIMIT)

SELECT SMALL SUCH THAT (X*Y < LIMIT) (*EXAMPLE*)
```

Examples of structures including SELECT nodes are:

```
R_NET RADAR_RETURN.
    STRUCTURE
        INPUT_INTERFACE RETURN_BUFFER
        ALPHA CHECK_VALIDITY
        SELECT ENTITY_CLASS OBJECT SUCH THAT
            (RANGE < PERIMETER_RANGE + DELTA)
        SUBNET UPDATE_THREAT_ESTIMATE
    END.

SUBNET UPDATE_THREAT_ESTIMATE.
    STRUCTURE
        ALPHA RECORD_UPDATE (*KEEP A RUNNING ACCOUNT*)
        SELECT OBJECT_THREATENING SUCH THAT
            ((XDOT*XDOT > XDOTLIM) AND (NOT DECOY))
            (*IGNORE DECOYS*)
    END.
```

Formally, the syntax for a SELECT node is:

```
<select node>::=

     SELECT {[ENTITY_CLASS] entity-class-name}1
            {[ENTITY_TYPE] entity-type-name  }1

          SUCH THAT <condition> [comment]

<condition>::=
     (<Boolean expression>)
```

Boolean Expression

A Boolean expression is in a form designed to mirror acceptable PDL 2 Boolean expressions with the exception that function references and set inclusion (IN) are not allowed. The Boolean expression is a rule of computation that, upon execution, produces a value of TRUE or FALSE. Some examples of typical Boolean expressions are:

```
X + 30 = Y
(YDOT <= 1500) = (XDOT >= 1800)
A OR ((NOT B) AND (NOT (C OR D)))
```

The complete formal definition of a Boolean expression is presented in Appendix D. The typical user is expected to have little or no reason to ever be concerned with this formal definition but should be aware of the following considerations:

1. If more than one operator occurs in a Boolean expression, the sequence of execution may be defined explicitly with parentheses or implicitly by the rule of operator precedence. Since the operator precedence may differ for different implementations of PDL 2 or PASCAL, the user is advised to fully parenthesize all Boolean expressions.

2. The appearance of an undefined name in a Boolean expression will result in an implicit definition of that name as an element of type DATA as discussed in Section 5.1.1.6.

3. If a data name has a value BOOLEAN for the attribute TYPE it will be treated as a <Boolean primary> (equivalent to a TRUE or FALSE). Otherwise, it will be treated as an <arithmetic factor> (equivalent to a number). The lack of a value BOOLEAN for the attribute TYPE in a context in which a <Boolean primary> is required will result in the detection of a syntax error by the RSL translator.

FOR EACH Node

A FOR EACH node consists of two physical nodes in a structure. The first node, termed the FOR EACH subject, is similar to an element node except that the element type name must be FILE, ENTITY_TYPE, or ENTITY_CLASS and the optional comment appears after the word DO in the syntax of the FOR EACH node. The second node, termed the FOR EACH body, is also similar to an element node, except that the element type name must be ALPHA or SUBNET. An optional comment follows the element name, and the entire FOR EACH body is bracketed by the words DO and END.

```
          ([FILE] file-name [RECORD]          )1
FOR EACH <[ENTITY_TYPE] entity-type-name      >   [SUCH THAT <condition>]
          ([ENTITY_CLASS] entity-class-name   )1

                ([ALPHA] alpha-name    )1
DO [comment]   <                        >  [comment] END
                ([SUBNET] subnet-name  )1
```

Examples of simple FOR EACH nodes are:

```
FOR EACH FILE OBJECTS_IN_TRACK
    DO ALPHA CHECK_RANGE END

FOR EACH ENTITY_TYPE THREAT
    DO (*LOOK AT ALL THREATS*) SUBNET CHECK_RANGE END
```

Two additional options are available on a FOR EACH node. First, the execution of the FOR EACH body may be made conditional based upon the value of some Boolean expression. This is accomplished by specifying the words SUCH THAT and a Boolean expression enclosed in parentheses immediately preceding the word DO. The second option is that if the FOR EACH subject names a FILE, the optional word RECORD may follow the file name.

Examples include:

```
FOR EACH FILE OBJECTS_IN_TRACK RECORD
    DO ALPHA CHECK_RANGE END

FOR EACH ENTITY_CLASS RANGE_MARK
    SUCH THAT (AMPLITUDE >= NOISE + AMP_DELTA)
    DO SUBNET BULK_FILTER (*REMOVE SLOW OBJECTS*) END
```

As with other nodes on a structure, the element type name may be omitted if the element is already defined. If the element has not been defined, the element type name must be supplied and the element is implicitly defined.

Formally, a FOR EACH node is defined as:

```
<for-each node>::=

         ([FILE] file-name [RECORD]             )1
FOR EACH <[ENTITY_TYPE] entity-type-name        >   [SUCH THAT <condition>]
         ([ENTITY_CLASS] entity-class-name      )1

                 ([ALPHA] alpha-name   )1
DO [comment]    <                      >   [comment] END
                 ([SUBNET] subnet-name )1
```

AND node

The AND node is a complex node with one entry arc and several exit arcs leading to parallel branches. There are two classes of AND nodes, rejoining and non-rejoining. In a rejoining AND node all of the branches rejoin at a virtual AND node with multiple entry arcs and a single exit arc. In a non-rejoining AND node, the branches do not rejoin; each of them ends with a TERMINATE, RETURN, or OUTPUT_INTERFACE node and there is no virtual AND node.

An AND node is written as the word DO, with an optional comment for the AND node, followed by two or more branches separated by the word AND; the entire construct is terminated with the word END. A branch is simply defined to be a sequence of one or more nodes. The syntax for an AND node is:

```
DO [comment] <branch> {AND <branch>}ⁿ₁ END
```

Some simple rejoining AND nodes include the following:

```
DO (*TWO BRANCH AND NODE*)
    ALPHA ONE
    SUBNET TWO
AND
    FOR EACH FILE THREE
        DO ALPHA FOUR END
END
DO (*THREE BRANCH AND NODE*)
    SELECT ENTITY_CLASS EC_1 SUCH THAT (X <= Y + 5.0)
    ALPHA CHECKER
    ALPHA DOER
AND
    SELECT ENTITY_CLASS EC_1 SUCH THAT (LARGE_BETA)
    SUBNET CHECK_R_DOT
```

```
AND
    ALPHA RECORD_HISTORY
END
```

An example of a non-rejoining AND node is the following:

```
DO (*BOTH BRANCHES TERMINATE*)
    ALPHA OUTPUT_HISTORY
    TERMINATE
AND
    SUBNET FORM_RADAR_REQUEST
    OUTPUT_INTERFACE RADAR_ORDERS
END
```

Since a branch is simply a sequence of nodes, AND nodes may be embedded within other AND nodes to any level desired. The following example embeds an AND node within the scope of another AND node.

```
DO (*ILLUSTRATE EMBEDDING*)
    ALPHA CHECKER
    DO (*EMBEDDED AND NODE*)
        ALPHA DOER
    AND
        SUBNET CHECK_RANGE
    AND
        ALPHA UPDATE_HISTORY
    END
AND
    SUBNET SELECT_LARGEST
END
```

Formally, the syntax for an AND node is:

<and node>::=

DO [comment] <branch> $\left\{\text{AND <branch>}\right\}_1^n$ END

<branch>::=

$\left\{\text{<node>}\right\}_1^n$

Rules Regarding AND Nodes

The following rules are enforced for AND nodes.

1. Each branch of an AND node must end in a TERMINATE, RETURN, or OUTPUT_INTERFACE or they all must rejoin. A mixture of terminating and non-terminating branches is not allowed.

2. There must be at least two branches, i.e., the word AND must appear at least once.

3. The words DO and END completely bracket the AND node.

4. No node in a structure may follow a non-rejoining AND node since all branches in a non-rejoining AND node terminate.

OR Node

Like the AND node, the OR node is a complex node with a single entry arc and multiple exit arcs. The meaning, however, is that only one of the exit branches is to be followed. The branch to be followed is determined by the value of Boolean expressions given for each branch. The order in which the Boolean expressions are to be evaluated may be specified by assigning ranking ordinals to the branches. In the absence of ordinals, the lexical ordering of the input text is used to determine the evaluation order. To cover the case where all Boolean expressions may evaluate to false, an otherwise clause is used.

Again, like the AND node, the OR node may be rejoining or non-rejoining. If the OR node is rejoining, all the branches rejoin at a virtual OR node. If the OR node is non-rejoining, each branch ends with a TERMINATE or RETURN node or OUTPUT_INTERFACE element node and there is no virtual OR node.

An OR node is written as the word IF, with an optional comment for the OR node, followed by one or more conditional branches separated by the word OR, in turn followed by the word OTHERWISE, an optional branch, and the word END. A conditional branch consists of an optional unsigned integer representing an ordinal for the branch, followed by a Boolean expression enclosed in parentheses and a sequence of one or more nodes. The syntax for an OR node is:

$$\text{IF [comment] [unsigned-integer] (<Boolean expression>)} \left\{\text{<node>}\right\}_1^n$$

$$\left\{ \text{OR [unsigned-integer] (<Boolean expression>)} \left\{\text{<node>}\right\}_1^n \right\}_0^n$$

$$\text{OTHERWISE} \left\{\text{<node>}\right\}_0^n \text{ END}$$

An example of a rejoining OR node not using ordinals is the following:

```
IF (*TWO BRANCHES*)
    (X > Y) ALPHA X_LARGER
OR
    (X < Y) ALPHA Y_LARGER
OTHERWISE
    ALPHA X_Y_EQUAL
END
```

An example of a non-rejoining OR node using ordinals is the following:

```
IF (*TWO BRANCHES, EMPTY OTHERWISE*)
    2 (RANGE <= FAR_LIMIT)
    ALPHA CONSIDER_OBJECT
    SUBNET UPDATE_STATE
    TERMINATE
OR
    1 (RANGE >= NEAR_LIMIT)
    SUBNET DROP_OBJECT
    TERMINATE
OTHERWISE
    TERMINATE
END
```

Like AND nodes, OR nodes may be nested to any level. For example, the following illustrates an OR node with a nested OR node.

```
IF
    ((X + Y) = 2 + 30)
    IF (*NESTED OR node*)
        (X > Z + 5)
        ALPHA ONE
        ALPHA TWO
    OR
        (Z - 10 < 25)
        ALPHA THREE
    OTHERWISE
    END
```

```
OR (X > Y)
   ALPHA FOUR
OTHERWISE
END
```

Formally the syntax for an OR node is:

<or node>::=

IF [comment] <conditional branch> $\left\{\text{OR <conditional branch>}\right\}_0^n$

OTHERWISE [<branch>] END

<conditional branch>::=

[unsigned integer] <condition> <branch>

<branch>::=

$\left\{\text{<node>}\right\}_1^n$

Rules Regarding OR Nodes

The following rules regarding OR nodes are enforced.

1. Each branch of an OR node must end in a TERMINATE, RETURN, or OUTPUT_INTERFACE or they all must rejoin. A mixture of terminating and non-terminating branches is not allowed.
2. There must be at least one branch plus an otherwise clause, i.e., the word OR is not required to appear but the word OTHERWISE is required.
3. Ordinals must be unsigned integers between 1 and 9999 inclusive.
4. No two branches can have the same ordinal.
5. Either all branches (except the OTHERWISE) must have ordinals or none may have ordinals. A mixture of ordinal and non-ordinal branches is not allowed.
6. The words IF and END completely bracket the OR node.
7. No node may follow a non-rejoining OR node in a structure since all branches in a non-rejoining OR node terminate.

CONSIDER OR Node

The CONSIDER OR node, like the OR node, is a complex node with a single entry arc and multiple exit arcs. The intended meaning is that only one of the exit branches is to be followed; the correct branch being determined by the element name being considered, termed the consider-subject, and the names specified in the conditional expressions for each branch.

Again, like the OR node, the CONSIDER OR node may be rejoining or non-rejoining. If the CONSIDER OR node is rejoining, all the branches rejoin at a virtual OR node. If the CONSIDER OR node is non-rejoining, each branch ends with a TERMINATE, RETURN, or OUTPUT_INTERFACE node and there is no virtual OR node.

There are two types of CONSIDER OR nodes, the consider-data and the consider-entity-class; distinguished by the element type of the element used as the consider-subject. For either type, the CONSIDER OR node is written in the same manner: the word CONSIDER, an optional element type name DATA or ENTITY_CLASS and a data or entity class name, followed by the word IF with an optional comment, followed in turn by two or more consider branches separated by the word OR, followed by the word END. Each consider branch consists of a consider list enclosed in parentheses followed by zero or more nodes comprising the branch. The form of the consider list is a sequence of one or more names separated by the word OR. If the consider-subject is a data name, these names are assumed to be values in its RANGE. If the consider-subject is an entity class name, these names are assumed to be entity type names and implicit declarations will result if they have not been previously defined.

The syntax for a consider-data node is:

CONSIDER [DATA] enumerated-data-name

$$\text{IF [comment]} \left(\text{enumeration-value-name} \left\{ \text{OR enumeration-value-name} \right\}_0^n \right)$$

$$\left\{ \text{<node>} \right\}_0^n$$

$$\left\{ \text{OR} \left(\text{enumeration-value-name} \left\{ \text{OR enumeration-value-name} \right\}_0^n \right) \left\{ \text{<node>} \right\}_0^n \right\}_1^n$$

END

The syntax for a consider-entity-class node is:

CONSIDER [ENTITY_CLASS] entity-class-name

IF [comment] $\left(\text{entity-type-name} \left\{\text{OR entity-type-name}\right\}_{0}^{n}\right) \left\{\text{<node>}\right\}_{0}^{n}$

$\left\{\text{OR} \left(\text{entity-type-name} \left\{\text{OR entity-type-name}\right\}_{0}^{n}\right) \left\{\text{<node>}\right\}_{0}^{n}\right\}_{1}^{n}$

END

Examples of a CONSIDER OR node are the following:

```
CONSIDER DATA OBJECT_CLASSIFICATION
IF (*BRANCH ON CLASS*)
    (UNKNOWN OR RV)
        SUBNET CHECK_BETA
OR
    (DECOY)
        SUBNET CHECK_BETA
        ALPHA UPDATE_WEIGHT
OR
    (JUNK)
END
CONSIDER ENTITY_CLASS OBJECT
IF (*BRANCH ON AGE OF OBJECT*)
    (ANCIENT OR VERY_OLD OR OLD)
    ALPHA UPDATE_AGE
    TERMINATE
OR
    (YOUNG)
    ALPHA CHECK_CHARACTERISTICS
    SUBNET UPDATE_STATE
    OUTPUT_INTERFACE TRACK_REQUEST
OR
    (INFANT)
    SUBNET UPDATE_STATE
    OUTPUT_INTERFACE TRACK_REQUEST
END
```

Differences Between OR Nodes and CONSIDER OR Nodes

Although there are many similarities between OR nodes and CONSIDER OR nodes, the following differences should be kept in mind.

1. An OTHERWISE branch is required on an OR node and not allowed on a CONSIDER OR node.
2. The only branch which may be empty on an OR node is the OTHERWISE branch. Any one of the branches on a CONSIDER OR node may be empty (but only one).
3. The branch conditions for an OR node must be Boolean expressions. For a CONSIDER OR node, the branch condition must be one or more value names or entity type names separated by the word OR.
4. Ordinals are optional on OR node branches and not allowed on CONSIDER OR node branches.

Formally, the syntax for a CONSIDER OR node is:

```
<consider-or node>::=
        <consider-data>
      | <consider-entity-class>

<consider-data>::=
        CONSIDER [DATA] enumerated-data-name
        IF [comment] <consider-data branch>
        {OR <consider-data branch>}^n_1
        END

<consider-data-branch>::=
        (enumeration-value-name {OR enumeration-value name}^n_0) [<branch>]

<consider-entity-class>::=
        CONSIDER [ENTITY_CLASS] entity-class-name
        IF [comment]  consider-entity-branch
        {OR <consider-entity-branch>}^n_1
        END

<consider-entity-branch>::=
        (entity-type-name {OR entity-type-name}^n_0 )[<branch>]
```

Rules Regarding CONSIDER OR Nodes

The following rules are enforced for *CONSIDER OR nodes*.

1. The consider-subject must be of type DATA or ENTITY_CLASS.
2. If the consider-subject is of type DATA, it must have the value ENUMERATION for attribute TYPE or it must have no value for attribute TYPE.
3. For a consider-data, the names in the consider-list must not be known names. (Note: The RSL translator has no knowledge of names which were entered inside of text strings, such as the value for attribute RANGE.) They are assumed to be value names contained within the RANGE for the data.
4. For a consider-entity-class, if the names in the consider-list are known names, they must be entity type names.
5. A CONSIDER OR node may have at most one empty branch and must have at least one non-empty branch.
6. Each branch of a CONSIDER OR node must end in a TERMINATE **or** RETURN node or OUTPUT_INTERFACE element node or they all must rejoin. A mixture of terminating and non-terminating branches is not allowed.

5.1.1.5 Declaring A Path

A path may be declared for any RSL element which is of type VALIDATION_PATH, by specifying a path declaration optionally preceded by the word INSERT. The path declaration itself consists of the word PATH followed by one or more element nodes, the word END, and an optional comment. Formally, this is shown as:

$$\text{[INSERT] PATH } \left\{\text{<element node>}\right\}_1^n \text{ END [comment].}$$

Each element node consists of an element type name followed by an element name and an optional comment. If the element has been previously defined, the element type name may be omitted. If the element has not been defined, the element type name must be specified and the element is implicitly defined (see Section 5.1.1.6). In either case, the comment is associated with the path node and not with the element itself.

Element nodes on a path may be constructed only from elements of a type defined with STRUCTURE APPLICABILITY PATH. The types so defined are EVENT and VALIDATION_POINT.

Examples of path declarations include the following:

```
VALIDATION_PATH VP_1.
    PATH
        EVENT TRIGGER_R_NET_ONE
        VALIDATION_POINT RECORD_DATA
        EVENT TRIGGER_R_NET_TWO
    END.

VALIDATION_PATH CHECK_TIMING.
    PATH
        VALIDATION_POINT RECORD_RADAR_RETURNS
        EVENT SCHEDULE_PULSES (*ENABLE SCHEDULER*)
        VALIDATION_POINT RECORD_RADAR_ORDERS
    END (*CHECK TIMING*).
```

The syntax for declaring a path is:

[INSERT] <path declaration>

<path declaration>::=

PATH $\left\{\text{<element node>}\right\}_1^n$ END [comment].

<element node>::=

[element-type-name] element-name [comment]

5.1.1.6 Implicit Element Declarations

An element may be declared either explicitly, by being the subject element of an element definition, or implicitly, by being referenced as part of a relationship, structure, or path declaration. For example, the element definition:

```
DEFINE R_NET R1.
    ENABLED BY EVENT E1.
    STRUCTURE
        ALPHA A1
        TERMINATE
    END.
```

implies the element declarations:

```
DEFINE EVENT E1.
DEFINE ALPHA A1.
```

With two exceptions, an element type name must be explicitly stated in order for an implied element declaration to be allowed. For example, if E1 and A1 had not been previously declared, then the above example would be legal, but the following would not be legal:

```
DEFINE R_NET R1.
    ENABLED BY E1.
    STRUCTURE
        A1
        TERMINATE
    END.
```

Note that an element type name applies only to the element name that immediately follows it. For example, in the following, DATA applies only to DAT1 and not to DAT2 or DAT3:

```
ALPHA AL1.
    INPUTS DATA DAT1, DAT2, DAT3.
```

The two exceptions to the rule that an undefined element name must always be preceded by an element type name occur in Boolean expressions and consider-lists. In a Boolean expression, undefined names encountered are implicitly defined to be of type DATA; the element type name DATA must not appear. In a consider-list for a consider-entity type of CONSIDER OR node, undefined names encountered are implicitly defined to be of type ENTITY_TYPE; the element type name ENTITY_TYPE must not appear.

5.1.2 Modifying an Element Definition

Once an element has been defined, it is often necessary to refine or modify that definition in some way. These modifications include the insertion of new declarations and the removal of existing declarations. Each of these types of modifications as they relate to the RSL concepts of elements, attributes, relationships, structures, and paths are discussed below. In several cases the material would basically be a restatement of material presented in previous sections of this document; the user will be referred to those sections.

The format used to modify the definition of an element closely follows that used to define a new element. First a declaration must be given of the subject element which is to be modified. This declaration is followed by the declarations of the desired changes for the subject element. The syntax for an element modification is:

[MODIFY] element-type-name element-name [comment].

$$\left\{\begin{array}{l}\text{[INSERT] <element definition sentence>}\\ \text{<attribute declaration removal>}\\ \text{<relationship declaration removal>}\\ \text{<structure declaration removal>}\\ \text{<path declaration removal>}\end{array}\right\}_0^n$$

As shown in the syntax, the word MODIFY is optional in an element modification. If the element to be modified exists in the ASSM then MODIFY will be assumed and need not be stated. If, however, the element is not defined in the ASSM, the RSL function assumes that this is a new element definition. Because of this assumption, the user is safest if he always uses the word MODIFY when modifying an element.

The word INSERT is also optional before an element definition sentence. Here, the use or omission of the word INSERT has no effect on the interpretation of the input.

5.1.2.1 Declaring the Element to be Modified

The declaration of the subject element to be modified is given by optionally specifying the word MODIFY, followed by the element type name, element name, and optional comment for the element.

[MODIFY] element-type-name element-name [comment].

The main purpose of this declaration is to specify which element is to be modified in the following declarations. It also may serve to modify the element definition itself since,if a comment is specified, it will replace any existing comment associated with the element. If no comment is specified, any existing comment for the element will be retained. For example, the first example below retains any existing comment for AL1 while the second associates the comment "(*NEW COMMENT*)" with the element AL2.

```
MODIFY ALPHA AL1.
MODIFY ALPHA AL2  (*NEW COMMENT*).
```

The syntax used to declare the element to be modified is:

```
<element modification>::=
    [MODIFY] element-type-name element-name [comment].
```

5.1.2.2 Declaring an Attribute Value

There is no difference between declaring attribute values for an existing element and declaring attribute values for a new element. The reader is referred to Section 5.1.1.2 above for a discussion of declaring attribute values.

5.1.2.3 Removing an Attribute Value

An existing attribute value for an element is removed by specifying the word REMOVE followed by the name of the attribute. Notice that the attribute value itself is not specified, nor is a comment.

```
REMOVE attribute-name.
```

The following example first defines an element DAT1 with several attributes, then the element definition is modified to remove attributes USE and INITIAL_VALUE, other attributes for DAT1 remaining intact. A new value for attribute INITIAL_VALUE is then declared.

```
DEFINE DATA DAT1.
    TYPE INTEGER.
    USE GAMMA.
    INITIAL_VALUE 0 (*NOMINAL VALUE*).
    MAXIMUM_VALUE 100.
MODIFY DATA DAT1.
    REMOVE USE.
    REMOVE INITIAL_VALUE.
    INSERT INITIAL_VALUE -1.
```

Rules Regarding Removal of Attribute Values

The following rules are enforced on removal of attribute values.

1. The word REMOVE must be specified.
2. The attribute value must not be specified.
3. The removal of an attribute value also removes any comment associated with the attribute value for the element.

4. If a comment is specified after the attribute name, an informative error diagnostic will be generated and the comment ignored. The removal will, however, still be accomplished.

Formally, the syntax for removing an attribute value is:

```
<attribute declaration removal>::=
    REMOVE attribute-name.
```

5.1.2.4 Declaring a Relationship Instance

There is no difference between declaring a relationship instance for an existing element and declaring a relationship instance for a new element. The reader is referred to Section 5.1.1.3 above for a discussion of declaring relationship instances.

5.1.2.5 Removing a Relationship Instance

A relationship between the subject element and one or more object elements is removed by specifying the word REMOVE, the relationship name optionally followed by the appropriate relation optional word for the relationship, followed by the names of the object elements. Each of these object element names may be preceded by its element type name.

REMOVE relation-name [relation-optional-word]

$\left\{ \text{[element-type-name] element-name} \right\}_1^n .$

Assume that DAT2 is defined as follows:

```
DEFINE DATA DAT2 (*DUMMY EXAMPLE*).
    INPUT TO ALPHA AL2 (*DAT2 - AL2*),
             ALPHA AL21,
             ALPHA AL22.
    INCLUDES DATA DAT21,
             DATA DAT22 (*DAT2 - DAT22*).
```

The following will remove the relationship INPUT between DAT2 and both AL21 and AL2. It will also remove the relationship INCLUDES between DAT2 and DAT22.

```
MODIFY DATA DAT2.
    REMOVE INPUT AL2, ALPHA AL21.
    REMOVE INCLUDES DATA DAT22.
```

Since a relationship instance relates a subject to an object, it is always possible to remove the relationship instance from either viewpoint. The following modifications accomplish exactly the same effect as those above.

```
MODIFY ALPHA AL2.
    REMOVE INPUTS DATA DAT2.
MODIFY ALPHA AL21.
    REMOVE INPUTS DAT2.
MODIFY DATA DAT22.
    REMOVES INCLUDED IN DAT2.
```

The net effect of either set of modifications is to leave DAT2 as if it were defined:

```
DATA DAT2 (*DUMMY EXAMPLE*).
    INPUT TO ALPHA AL22.
    INCLUDES DATA DAT21.
```

The formal syntax for the removal of relationship instances is:

<relationship declaration removal>::=

REMOVE relation-name [relation-optional-word]

$\left\{\text{[element-type-name] element-name}\right\}_1^n$.

Rules Regarding Removal of Relationship Instances

The following rules are enforced on removal of relationship instances.

1. The word REMOVE must be specified.
2. The removal of a relationship instance also removes any comment associated with that instance.
3. If a comment is specified after an element name, an informative error diagnostic will be generated and the comment ignored. The removal, however, will still be accomplished.

5.1.2.6 Declaring a Net Structure

There is no difference between declaring a structure for an existing element and declaring a structure for a new element. The reader is referred to Section 5.1.1.4 above for a discussion of declaring a structure for an element.

5.1.2.7 Removing a Net Structure

A structure for an element is removed by specifying the word REMOVE followed by the word STRUCTURE. For example, if SUB1 is defined as follows:

```
DEFINE SUBNET SUB1.
    STRUCTURE
        FOR EACH ENTITY_CLASS EC1
            DO SUBNET SUB2 END
        DO ALPHA AL1
        AND ALPHA AL2
        AND ALPHA AL3
        END
        RETURN
    END (*ARBITRARY STRUCTURE*).
```

the text:

```
MODIFY SUBNET SUB1.
    REMOVE STRUCTURE.
```

will remove the structure associated with SUB1 and its comment.

Formally, the syntax for removal of a structure is:

```
<structure declaration removal>::=
    REMOVE STRUCTURE.
```

Rules Regarding Removal of Structures

The following rules are enforced for removal of structures.

1. The word REMOVE must be specified.
2. The removal of a structure also removes any comment associated with the structure.
3. If a comment is specified after the word STRUCTURE, an informative error diagnostic will be generated and the comment ignored. The removal of the structure will still be accomplished.

5.1.2.8 Declaring a Path

Declaring a path for an existing element is no different from declaring a path for a new element. The reader is referred to Section 5.1.1.5 for a discussion of declaring a path for an element.

5.1.2.9 Removing a Path

A path for an element is removed by specifying the word REMOVE followed by the word PATH. For example, if VAL_1 is defined as follows:

```
DEFINE VALIDATION_PATH VAL_1.
    PATH
        VALIDATION_POINT VP_ONE
        EVENT EVT_ONE
        VALIDATION_POINT VP_TWO
    END (*DUMMY PATH*).
```

the text:

```
MODIFY VALIDATION_PATH VAL_1.
    REMOVE PATH.
```

will remove the path associated with VAL_1 and its comment.

Formally, the syntax for removal of a path is:

```
<path declaration removal>::=
    REMOVE PATH.
```

Rules Regarding Removal of Paths

The following rules are enforced by the RSL translator for removal of paths.

1. The word REMOVE must be specified.
2. The removal of a path also removes any comment associated with the path.
3. If a comment is specified after the word PATH, an informative error diagnostic will be generated and the comment ignored. The removal of the path is still accomplished.

5.1.3 Deleting an Element

An element may be deleted only if it has no relationships to any other elements, has no associated structure or path, and is not referenced on any structure or path. An element is said to be referenced on a structure or path if the element name is used in any of the following contexts in a structure or path:

1) as an element node

2) within a Boolean expression

3) within a consider-entity-type list

4) as a consider-subject

5) as a for-each-subject

6) as a for-each-object

7) as a select-subject

If an element has values for one or more attributes, it is not necessary to remove these values before deleting the element. If an element has relationships to other elements, these relationships must be removed by modifying either the element itself or the elements to which it is related before the element may be deleted. If an element has a path or structure, that path or structure must be removed by modifying the element before the element may be deleted. If an element is referenced on a path or structure, either the path or structure must be changed using the Interactive R-Net Generation Function (see Section 5.2) to remove the reference or the element with which the path or structure is associated must be modified to remove the path or structure, before the element may be deleted.

Once the necessary modifications have been performed, the element is deleted by specifying the word DELETE followed by the element type name and element name. Since this sentence results in the complete removal of all information about the element deleted, it is meaningless to follow it with any attribute, relationship, path, or structure declarations.

An element defined as:

```
DEFINE ALPHA DUMMY (*DUMMY COMMENT*).
```

may be simply deleted by:

```
DELETE ALPHA DUMMY.
```

Formally, the syntax for deleting an element is:

```
<element deletion>::=
    DELETE element-type-name element-name.
```

A comment may be specified following the element name but it will result in an informative diagnostic and the comment will be ignored.

5.1.3.1 Deleting an Element with Attribute Values

No special action need be taken to remove attribute values for an element before the element may be deleted. This is true because attributes are considered to be strictly local to an element, i.e., they affect no other element, and therefore their automatic removal on deletion of the element does not in any way affect the definition of any other element. The user is, of course, always free to explicitly remove any or all attributes of the element before deleting it. Section 5.1.2.3 of this document discussed the process of removing attribute values for an element.

For example, if DATA XYZ is defined as:

```
DEFINE DATA XYZ (*XYZ COMMENT*).
    DESCRIPTION "STRICTLY FOR AN EXAMPLE".
    INITIAL_VALUE -37.315E-38.
    MAXIMUM_VALUE 0.0 (*MUST NOT BE > 0*).
```

then

```
MODIFY DATA XYZ.
    REMOVE DESCRIPTION.
    REMOVE INITIAL_VALUE.
    REMOVE MAXIMUM_VALUE.
DELETE DATA XYZ.
```

is exactly equivalent to:

```
DELETE DATA XYZ.
```

Both forms will completely remove XYZ, its comment, and all of its attribute values and their comments. The net result will be exactly the same as if DATA XYZ was never defined.

5.1.3.2 Deleting an Element with Relationships

Before an element may be deleted, all relationships between that element and all other elements must be explicitly removed. This may be accomplished by modifying the element to remove the relationships or by modifying the elements to which the element is related. The user is referred to Section 5.1.2.5 for a discussion of the removal of relationships between elements.

For example, if DATA XYZ is defined as follows:

```
DATA XYZ.
    INCLUDES DATA X
             DATA Y
             DATA Z.
    INPUT TO ALPHA ALF1.
```

then the INCLUDES and INPUT relationships must be removed before DATA XYZ may be deleted. The following modifies X, Y, Z and ALF1 to remove these relationships and then deletes XYZ.

```
MODIFY DATA X.
    REMOVE INCLUDED IN XYZ.
MODIFY DATA Y.
    REMOVE INCLUDED IN XYZ.
MODIFY DATA Z.
    REMOVE INCLUDED IN XYZ.
MODIFY ALPHA ALF1.
    REMOVE INPUTS XYZ.
DELETE DATA XYZ.
```

DATA XYZ could also have been modified directly to remove the relationships and then deleted:

```
MODIFY DATA XYZ.
    REMOVE INCLUDES DATA X.
    REMOVE INPUT TO ALF1.
    REMOVE INCLUDES Y,Z.
DELETE DATA XYZ.
```

In either case, the element XYZ, its comment and all relationships to other elements will be completely removed from the data base.

5.1.3.3 Deleting an Element with a Path or Structure

If an element has an associated path or structure, the path or structure must be removed before the element may be deleted. This may only be done by explicitly modifying the element to remove the path or structure as described in Sections 5.1.2.7 and 5.1.2.8 of this document or by using the RNETGEN function (see Section 5.2).

For example, suppose that SUB1 is defined:

```
SUBNET SUB1 (*HAS A STRUCTURE*).
    STRUCTURE
        CONSIDER DATA DAT1
            IF (*OR NODE*)
                (VAL 1 OR VAL2) ALPHA AL1
            OR
                (VAL3) ALPHA AL2
                       SUBNET SUB3
            END
        RETURN (*BACK TO CALLING STRUCTURE*)
    END.
```

Then, the following RSL commands would delete SUB1:

```
MODIFY SUBNET SUB1.
    REMOVE STRUCTURE.
DELETE SUBNET SUB1.
```

Again, suppose that VAL1 is defined as:

```
VALIDATION_PATH VAL1 (*HAS A PATH*).
    PATH
        EVENT EVT01
        EVENT EVT02
        OUTPUT_INTERFACE RECORDER
    END.
```

Then VAL1 may be deleted by entering:

```
MODIFY VALIDATION_PATH VAL1.
    REMOVE PATH.
DELETE VALIDATION_PATH VAL1.
```

5.1.3.4 <u>Deleting an Element Referenced on a Path or Structure</u>

As long as at least one path or structure contains a reference to a particular element, that element cannot be deleted. Therefore, in order to delete an element referenced on a path or structure, it is first necessary to remove that reference. A removal of the reference may be accomplished by using the Interactive R-Net Generation function to physically alter the structure or path, or by using RSL commands to remove the entire referencing structure or path. The use of the Interactive R-Net Generation function is detailed in Section 5.2.

In the example below, RSL commands are used to remove the structure and/or path which references the element to be deleted. Assume R_NET NET01 and VALIDATION_PATH VALPATH_01 are defined as:

```
R_NET NET01.
    STRUCTURE
        EVENT EVT01 (*TRIGGER NET02*)
        ALPHA ALF1
        VALIDATION_POINT VALPOINT_01
        TERMINATE
    END.
VALIDATION_PATH VALPATH_01.
    PATH
        EVENT EVT00
        EVENT EVT01
        VALIDATION_POINT VALPOINT_01
    END.
```

If the deletion of ALF1 is desired then only the structure for NET01 must be removed:

```
MODIFY R_NET NET01.
    REMOVE STRUCTURE.
DELETE ALPHA ALF1.
```

If the deletion of EVT01 or VALPOINT_01 were desired then both the structure and the path would have to be removed. For example:

```
MODIFY R_NET NET01.
    REMOVE STRUCTURE.
MODIFY VALIDATION_PATH VALPATH_01.
    REMOVE PATH.
DELETE VALIDATION_POINT VALPOINT_01.
```

5.1.4 Renaming an Element

The name of any element may be changed to a new name as long as that new name is not currently in use. The form of the statement is the word RENAME followed by the old element name, the word AS, the new element name and an optional comment. Any existing comment for the element will be removed. If a new comment is specified it will be associated with the new element name.

For example, assume that OLD_ELT was declared as:

```
ALPHA OLD_ELT (*COMMENT FOR OLD_ELT*).
```

then

```
RENAME OLD_ELT AS NEW_ELT.
```

will result in the following information content in the data base:

```
ALPHA NEW_ELT.
```

The rename command is equivalent to a modification of an existing element definition. Thus the rename command should only be specified between element definitions, not within the declarations of any element definition.

Formally, the syntax for a rename is:

```
<RSL command>::=
    RENAME element-name AS new-element-name [comment].
```

Cautions for Renaming

The user should be aware that renaming an element will not change the name in material which is stored in the data base in the form of character strings. The following are stored in this form:

1. All comments, i.e., material enclosed within the comment brackets (* and *).

2. All text strings, i.e., material enclosed within the text brackets " and ". Note that this includes all code within the executable description (BETA or GAMMA attribute value) for an ALPHA as well as the code within a TEST attribute value for a PERFORMANCE_REQUIREMENT.

3. All conditionals in structures, i.e., material enclosed within the conditional brackets (and).

5.1.5 Retyping an Element

The element type of any element may be changed as long as the new type is compatible with all current uses of the element. This means that the following must all be true:

1. All attributes for the element must be legal for the new element type.
2. All relationships the element has must be legal for the new element type.
3. All references to the element on structures and paths must be in a context which is allowable for the new element type.
4. The element must not have an associated structure or path.

An element is retyped by specifying the word RETYPE, the element name, the word AS, and the new element type name. Like a rename, the retype command is equivalent to a modification of an existing element definition and should only be specified between element definitions, not within the declarations of any element definition.

Assume that the following element definitions exist:

```
ALPHA AL (*THIS IS ALPHA AL*).
    DESCRIPTION "GOOD_OLD AL".
DATA DAT1.
    INITIAL_VALUE -5.0.
SUBNET SUB1 (*COMMENT FOR SUB1*).
    STRUCTURE
        ALPHA X
        SUBNET Y
        RETURN
    END.
```

The following retype would be acceptable since the attribute DESCRIPTION is legal for DATA.

```
RETYPE AL AS SUBNET.
```

The resulting definition of AL is:

```
SUBNET AL (*THIS IS ALPHA AL*).
    DESCRIPTION "GOOD_OLD AL".
```

The following two retypes would not be acceptable; the first because a FILE cannot have an INITIAL_VALUE and the second because SUB1 has an associated structure.

```
RETYPE DAT1 AS FILE.
RETYPE SUB1 AS R_NET.
```

Formally, the syntax for a retype is:

```
<RSL command>::=
    RETYPE element-name AS element-type-name.
```

5.1.6 Using Synonyms

The user may define an alternate name for an element by introducing another element of type SYNONYM and the relationship EQUATES between the two elements. A synonym name EQUATES to at most one element name, termed the prime name. However, a single element name may be EQUATED to any number of synonym names. For example, the following EQUATES SYN1 and SYN2 to the prime name AL1:

```
DEFINE ALPHA AL1.
    EQUATED TO SYNONYM SYN1,
               SYNONYM SYN2.
```

Once a synonym name has been defined and EQUATES to a prime name, any reference to the synonym name is interpreted as a reference to the prime name. The one exception to this rule is when the element type name SYNONYM is stated explicitly.

If the intended reference is to the prime name, the element type name for the prime name should be used. Thus, given the definitions of AL1, SYN1 and SYN2 above, the following two structures are exactly equivalent.

```
SUBNET SUB1.
    STRUCTURE
        ALPHA SYN1
        SUBNET SUB2
        ALPHA AL1
        RETURN
    END.
```

```
SUBNET SUB1.
    STRUCTURE
        ALPHA AL1
        SUBNET SUB2
        ALPHA SYN2
        RETURN
    END.
```

To declare a synonym name or EQUATES relationship, or to remove such declarations, the element type name SYNONYM must be used. Given the above definitions of AL1, SYN1, and SYN2, the EQUATES relationships may be removed by stating.

```
MODIFY ALPHA A1.
    REMOVE EQUATED TO SYNONYM SYN1
                      SYNONYM SYN2.
```

The synonym element SYN1 and SYN2 may then be deleted:

```
DELETE SYNONYM SYN1.
DELETE SYNONYM SYN2.
```

Note that if the element type name SYNONYM had not been stated explicitly, the modification above would have been interpreted as:

```
MODIFY ALPHA A1.
    REMOVE EQUATED TO ALPHA A1
                      ALPHA A1.
```

which is illegal.

Cautions on the Use of Synonyms

Synonyms should not be used in any material which is stored in the data base in the form of character strings. The following are stored in this form:

1. Comments, i.e., material enclosed within (* and *).

2. Text strings, i.e., material enclosed within double quotes. This includes all BETA and GAMMA attribute values for ALPHAs and TEST attribute values for PERFORMANCE_REQUIREMENTs.

3. Conditionals in structures, i.e., material enclosed within (and).

5.2 GENERATING STRUCTURE GRAPHICS INTERACTIVELY (RNETGEN FUNCTION)

This section describes the Interactive R-Net Generation function (RNETGEN) which provides an interactive (prompting) graphics method of creating and storing into the ASSM the structures for R_NETs, SUBNETs, and VALIDATION_PATHs. This interactive graphics facility is only available when REVS is operating in the on-line mode (see Section 4.3) and uses the ANAGRAPH color graphics terminal for displaying and manipulating a graphical representation of such structures. Figure 5-1 presents a typical SUBNET structure as displayed on the screen.

The Interactive R-Net Generation function is invoked by issuing the RCL statement, RNETGEN. REVS must be in the on-line mode prior to issuing this RCL statement. Should the RNETGEN statement be issued in the off-line mode, an appropriate diagnostic message will be generated in the REVS.OUT file, and processing will continue with the next RCL statement. Upon successful processing of the RNETGEN statement, the message "ENTER TRACKBALL TO CONTINUE" is displayed on the screen.* The user must respond by depressing the trackball entry key in order for RNETGEN to continue. The screen is subsequently erased and redrawn with a menu appearing along its left side as depicted in Figure 5-1.

This menu list consists of the basic operations provided by the RNETGEN function. Any one of these operations is invoked by selecting the appropriate menu line entry via the trackball input facility (i.e., the trackball cursor is positioned on the menu line and the trackball entry key depressed). The last selected menu line remains in force (active) until overridden by a subsequent menu selection; the white bullet marker appearing at the left edge of each menu line is colored red for the currently active menu selection.

Depending upon the current status of RNETGEN, the selection of a particular operation may be illegal and result in the message "ILLEGAL MENU SELECTION". For example, prior to building or retrieving a structure, user selection of the MOVE NODE menu operation would be illegal. In such cases the diagnostic message is displayed, the input request is ignored, and some other menu operation must be selected by the user.

*All of the messages displayed by RNETGEN are listed and described in Appendix E.

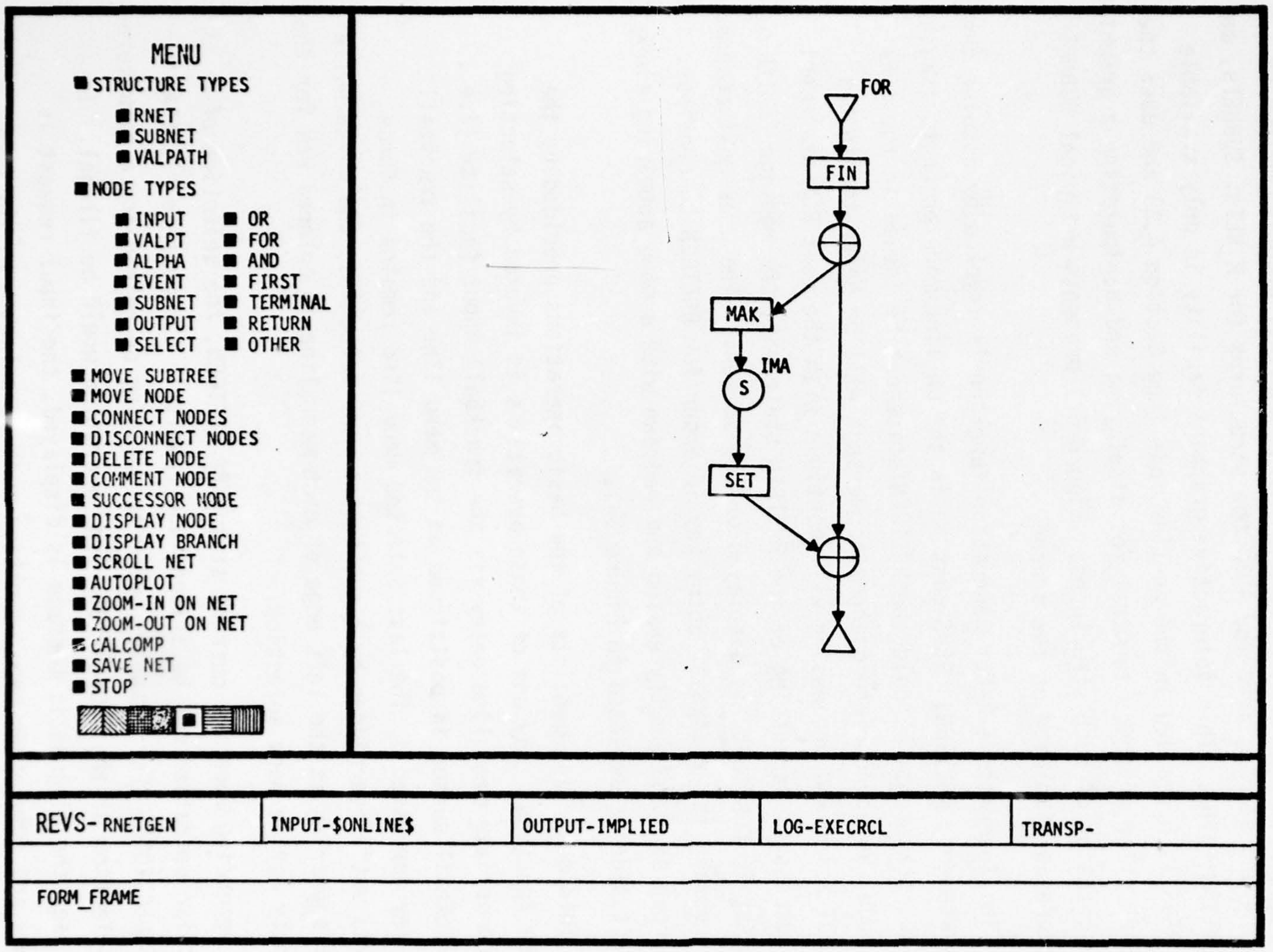


Figure 5-1 RNETGEN Display

The color menu appearing below the menu list provides the capability for specifying a color, of which there are seven, to be used in displaying nodes on the structure. A more detailed description of color selection is given in Section 5.2.2.8. The currently active color selection is indicated with either a small solid or outlined black square located in the center of the selected color square. Turquoise is the default color selection provided by RNETGEN initiation.

5.2.1 Identifying the Structure

There are three types of structures which may be created and stored or retrieved and modified by the RNETGEN function. As indicated in the menu, these are R_NET, SUBNET, and VALIDATION_PATH structures. Upon selection of one of these menu entries, the message "KEYIN ELEMENT NAME OF DESIRED ELEMENT" is displayed just below the color menu display. The user responds to this message by typing in the name of the element to which the desired structure is (or is to be) attached. If the name is illegal (i.e., the name does not begin with an alphabetic character and contains characters other than alphanumeric or the underscore), then the message "ILLEGAL SYNTAX, INPUT REJECTED" is displayed and the input is rejected. If the name is already in use in the ASSM, but for some other purpose, then an appropriate message is displayed and the user input is rejected. In either case, the user must re-issue a menu selection to continue.

However, if the input name already exists in the ASSM as an element of the appropriate type (R_NET, SUBNET, or VALIDATION_PATH), then the message "ELEMENT IN ASSM, IS IT THE ONE? KEYIN YES OR NO" is displayed. The user responds with a keyboard entry of YES (Y) or NO (N). If the response is NO then the input name is ignored and another menu selection must be issued to continue.

If the response is YES and a structure already exists in the ASSM, it will be copied to a temporary work area in the ASSM. The user will then be given the option to display the structure in either its zoomed-in or zoomed-out mode (see Section 5.2.3). The message "ZOOM-OUT OR ZOOM-IN ON STRUCTURE, SELECT VIA TRACKBALL" is displayed. The user responds by using the trackball to select the word ZOOM-OUT or the word ZOOM-IN in the displayed message. The structure will then be displayed accordingly.

If the structure contains no graphics coordinate data (i.e., the structure was created and stored in the ASSM via the RSL translator), then the user will be given the option to use either the autoplot capability or the prompting capability for entering graphics coordinate data on the structure. The message "NO GRAPHICS DATA ON STRUCTURE" is displayed followed by the message "SELECT - PROMPTER OR AUTOPLOT - VIA TRACKBALL". The user responds with a trackball selection of either PROMPTER or AUTOPLOT. If the autoplot mode is selected, all graphics coordinates for each node on the structure are generated automatically and the user is then given the option to display the resulting structure in either its zoomed-out or its zoomed-in mode. If the user selects the prompting mode for entering graphics coordinate data, then the entry node of the structure is displayed at the top center of the screen display area and the message "USE SUCCESSOR NODE MENU SELECTION FOR PROMPTING" is displayed below the menu list. A detailed description of this operation is given in Section 5.2.3.7.

If the ASSM element does not contain an associated structure, then an entry node for the structure is created and displayed at the top center of the display area. The user continues by making some other menu selection in order to add to the structure. The following section describes these operations in detail.

If the element input name does not exist in the ASSM the user will be informed by the message "ELEMENT NOT IN ASSM, DO YOU WANT IT ENTERED? KEYIN YES OR NO". If the response is NO then the input name is ignored and another menu selection must be issued to continue. If the user response is YES then the element name is entered in the ASSM and an entry node for its structure is created and displayed at the top center of the screen display area. Again, the user continues by making other menu selections in order to add to the structure.

5.2.2 Creating/Modifying a Structure

Once a structure or any portion thereof has been displayed on the screen in its zoomed-in mode, it may be altered through the use of any of the menu operations described in the following subsections. It should be pointed out, however, that any changes made are only temporary. Once the user has made all desired changes, the SAVE NET menu operation must be selected in order for the altered structure to become permanent in the ASSM.

5.2.2.1 Adding a Node

To add a node to the currently displayed structure the user must first select the desired node type from the list of available node types appearing in the menu. If the selected node type is illegal for the current structure, the message "ILLEGAL NODE FOR CURRENT STRUCTURE" will be displayed and the selection will be rejected. After successfully selecting a node type, the node is positioned in the structure display area. This is accomplished by selecting a point on the screen via the trackball.

One of two errors may result from this operation. If the selected point is too close to an already existing node, possibly causing a node overlap, the message "NODE OVERLAP IN STRUCTURE, REPEAT SELECTION SEQUENCE" will be displayed and another screen position must be selected by the user. Also, if the selected point is too close to the border of the screen display area, the message "NODE WILL NOT FIT ON DRAWING AREA, TRY AGAIN" may result and a new position will have to be selected.

If the selected node type references an element in the ASSM, the message "KEYIN DESIRED ELEMENT NAME" will be displayed; the user responds with a keyboard input of the appropriate name. If an input error is detected (i.e., a name syntax error or the element is already in the ASSM with inappropriate type), then the input is ignored, and the user must restart the input sequence by re-selecting the node screen position. If the element name is not in the ASSM, the user is given the option to either enter it or not enter it in the ASSM. This is done by keying in YES (Y) or NO (N) in response to the message "ELEMENT NOT IN ASSM, DO YOU WANT IT ENTERED? KEYIN YES OR NO".

If the selected node type is an OR node, the message "CONSIDER - DATA, ENTITY_CLASS, NEITHER - SELECT VIA TRACKBALL" is displayed and the user must respond by selecting one of the three entries in the message via the trackball. If the selection is either DATA or ENTITY_CLASS, the user will be required to enter the corresponding element name via the keyboard. Again, checks are made to ensure error-free input. If the supplied name does not exist in the ASSM, an option will be given to allow entry into the ASSM.

If the selected node type is a FOR EACH node, the message "FOR EACH - FILE, ENTITY_TYPE, OR ENTITY_CLASS - SELECT VIA TB" is displayed. The user must respond with a trackball selection of the appropriate element type from the list within the message. The corresponding element name is then keyed-in and similar processing is performed as with the OR node.

Once all required inputs have been supplied, error-free, the indicated node type symbol is displayed at the selected point on the screen. If the node references an element, the first three (3) characters of the element name are also displayed with the node symbol on the screen. Figure 5-2 gives a list of the available node types and their corresponding display symbols.

5.2.2.2 Removing a Node

Any node may be removed from the current structure by first selecting the DELETE NODE menu operation and subsequently using the trackball to point to the desired node on the displayed structure. If the message "NON-EXISTENT NODE SELECTED, RESTART SELECTION SEQUENCE" appears, the user has not correctly specified a node on the structure. This condition can happen if the trackball cursor is not properly positioned on the node, i.e., try to place it as close as possible to the node's center. When the node has been properly identified, it will be erased from the screen along with all of its connecting arcs. The corresponding information will also be removed from the ASSM and an appropriate message will be displayed to inform the user that the operation is complete.

5.2.2.3 Connecting Nodes

Node arcs are formed by first selecting the CONNECT NODES menu operation and subsequently using the trackball to indicate the predecessor node and its corresponding successor node. Checks are performed to determine if nodes do indeed exist at the selected points on the screen and, if so, whether the indicated nodes can be legally connected. If an error condition exists, the message "ILLEGAL NODE SELECTION, RESTART SELECTION SEQUENCE" is displayed. If either one of the selected nodes is found to be in error, both nodes must be re-selected in order to continue. If the selected

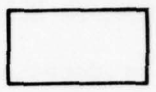
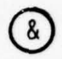
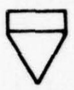
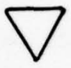
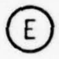
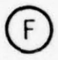
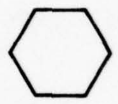
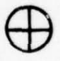
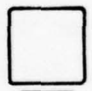
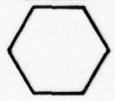
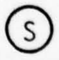
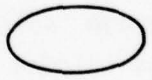
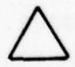
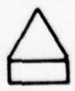
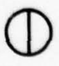
NODE TYPE		DISPLAY SYMBOL
ALPHA	-	
AND	-	
ENTRY NODE ON R_NET	-	
ENTRY NODE ON SUBNET	-	
EVENT	-	
FOR EACH	-	
INPUT_INTERFACE	-	
OR	-	
OTHER	-	
OUTPUT_INTERFACE	-	
SELECT	-	
SUBNET	-	
RETURN	-	
TERMINATE	-	
VALIDATION_POINT	-	

Figure 5-2 Node Display Symbols

predecessor node is an OR node and an implicit determination cannot be made as to its type (e.g., splitting or rejoining OR node), then the user must respond YES (Y) or NO (N) to the displayed message "IS PREDECESSOR NODE A REJOINING-OR-NODE, KEYIN YES OR NO".

If the predecessor node is a splitting CONSIDER OR node, then one of the two following messages will be displayed:

1) KEYIN ENTITY_TYPE ELEMENT(S) ASSOCIATED WITH THIS BRANCH ( ) or

2) KEYIN RANGE VALUE(S) ASSOCIATED WITH THIS BRANCH ( ).

The user responds by entering, via the keyboard, the requested branch information enclosed within parentheses. The branch conditions have the same syntax as RSL branch conditions (see Section 5.1.1.4). Syntax and semantic checks are performed on this input and if an error is detected, an appropriate diagnostic message is displayed and all inputs are ignored including node selections. Node selections must be re-issued in order to continue.

If the predecessor node is a splitting non-CONSIDER OR node and the branch requires an ordinal, then the message "KEYIN ORDINAL VALUE" is displayed. The user must then key in a four character (numeric) value to be attached to the indicated branch. Checks are performed on the input to insure correctness. If an error is detected (i.e., non-numeric value or duplicate ordinal), then an appropriate diagnostic is displayed and the ordinal value must be keyed in again. Once the ordinal value has been input without error or if no ordinal is required for the indicated branch, then the message "ENTER CONDITIONAL EXPRESSION SEGMENT" will be displayed. The user must respond by typing in the conditional expression enclosed within parentheses. If the conditional expression is the word OTHERWISE, then the parentheses are omitted on input.

If the predecessor node is a FOR EACH node, the user is given the option to supply a conditional expression to be attached to the branch.

After successfully supplying all input required to form the arc between the selected nodes, a white directed arc is displayed on the structure and the corresponding information is entered in the ASSM. An appropriate message is displayed to inform the user that the menu operation has been successfully completed.

5.2.2.4 Disconnecting Nodes

Node arcs are removed from the structure via the DISCONNECT NODES menu operation. This is accomplished by using the trackball to identify the two nodes on the structure which are to be disconnected. Error checks are performed to ensure that the selected nodes do indeed exist on the structure and that they do have arcs between them. If an error is detected, an appropriate message is displayed and the node selection sequence must be restarted. It should be pointed out that the node selection sequence is irrelevant (i.e., either the successor or predecessor node may be selected first). Upon successful identification of the two nodes, the node arc is erased from the screen and the corresponding information is removed from the ASSM. If the deleted branch was an OR or FOR EACH branch containing a conditional expression then this information is also removed from the ASSM. After successful completion of the operation, an appropriate message is displayed to inform the user that he may continue with other operations.

5.2.2.5 Commenting a Node

The COMMEND NODE menu operation is multipurpose in that it allows the user to enter, remove, or display comments on a node in the structure. After identifying the desired node using the trackball, the message "ENTER, REMOVE, OR DISPLAY COMMENT ON NODE, SELECT VIA TB" is displayed; the user responds with a trackball selection of the desired operation to be performed on the selected node.

If ENTER is selected, then the user is required to type in the desired comment enclosed within (* and *). Upon completion of this input, an appropriate message is displayed to inform the user that the operation has been performed.

If REMOVE is selected, the comment, if one exists, is detached from the selected node and deleted from the ASSM.

If DISPLAY is selected, the first line of the comment is displayed, if one exists, and the message "ENTER TRACKBALL TO CONTINUE" is displayed if there are more lines in the comment; otherwise, the message "ENTIRE COMMENT ON SELECTED NODE HAS BEEN DISPLAYED" is displayed. If the selected node is not commented, an appropriate message is displayed to so inform the user.

5.2.2.6 Moving a Node

The user may move any node appearing on the screen by selecting the MOVE NODE menu operation. The node to be moved is identified via a trackball selection. If the message "NON_EXISTENT NODE SELECTED, RESTART SELECTION SEQUENCE" appears, the user must re-issue the trackball node selection being more careful to locate the trackball cursor as near the node center as possible. Once the desired node has been successfully identified, the message "SELECT-TO POSITION-VIA TRACKBALL" is displayed; the user must respond using the trackball to select the point in the screen display area to which the selected node is to be moved.

This selection may also result in an error message. For example, if the selected point is too close to an existing node on the structure (i.e., it might result in a node overlap) or if the point is off the display area or too close to the display border (i.e., the entire node could not be displayed), an error condition exists and an appropriate diagnostic message is displayed. If this happens the selection sequence must be repeated beginning with the node selection.

After successfully identifying both the desired node and its new screen position, the node along with any arcs extending from it will be erased from its current location on the screen and will be redrawn with its associated arcs at the new position on the screen. The ASSM will be updated accordingly and a message will be displayed to inform the user that the operation has been successfully completed.

5.2.2.7 Moving a Subtree

This operation is similar to the moving of a node except that the entire portion of the structure below and including the selected node is moved. It should be pointed out that node overlapping will not be diagnosed in this operation as it is in the moving of a single node. Depending on the extent of the subtree, this operation will be somewhat slower to complete than any of the previously described operations. Once the operation is complete, an appropriate message is displayed to inform the user that he may continue further processing.

5.2.2.8 Selecting a Color

The color options available to the user appear as seven colored squares just below the last line in the menu list (see Figure 5-1). The selected color is used for displaying nodes on the screen as they are created by the user. It may also be used for changing a node color appearing on the displayed structure. The desired color is selected via the trackball prior to selecting the node type to be created or prior to selecting the node on the structure whose color is to be changed. So long as the selected color in the menu is identified by a small black outlined square, the node color change mode is in effect. However, as soon as any other menu selection is made, the black outlined square in the color menu becomes a **solid black** square indicating that a color menu selection is required to re-activate the node color change mode.

The selected color indicated in the menu is also used by the AUTOPLOT operation as described in Section 5.2.3.8 below.

5.2.2.9 Saving a Structure

As stated earlier, the structure currently being built or modified resides in a temporary work area of the ASSM. If the structure is to be given a permanent status in the ASSM the user must do so explicitly by selecting the SAVE NET menu operation. If this is not done prior to terminating or beginning on another structure then a warning message will be issued to inform the user that the current structure was not saved; the user may save the structure at this point.

Before the structure is actually saved, a structure analysis is performed to determine whether there are any errors existing in the structure. If an error is detected (i.e., an incomplete or incorrect structure), the structure is erased from the screen. It is redisplayed with the node in error centered in the display area with a red, white, and blue square surrounding it. An appropriate error message is also displayed explaining the cause of the error. At this point the user must correct the error before the structure can be saved.

Once the structure has been saved, and this may take several seconds depending on the size of the structure, a message will be displayed to inform the user that the structure was successfully saved.

5.2.3 Displaying a Structure and Its Elements

A structure may be displayed on the screen in one of two modes: zoomed-in mode or zoomed-out mode. As stated previously, structures can be created and modified only in the zoomed-in mode. The zoomed-out mode has been provided so that the user may view a structure in its entirety in a miniature color-coded form when the structure is too large to be displayed in its zoomed-in mode.

5.2.3.1 Scrolling a Structure

An additional feature has been provided to display any portion of large structures in the zoomed-in mode. This is accomplished using the SCROLL NET menu operation. Although this operation has been categorized as a display capability for structures or portions thereof, it would and does also fit in the category of functions used and required in the building and modifying of structures. It provides a windowing capability for a structure which is too large to fit on the screen in its zoomed-in mode.

Upon selection of this menu operation, the user selects a "from" and "to" position on the screen using the trackball. The "from" position indicates a point on the structure which is to be moved across the screen to the "to" position on the screen. Upon selection of the "from" position, a small blue solid circle is displayed serving as a reminder to the user of the selected point. Note that this point must be located in the screen display area; otherwise, a diagnostic message will be displayed and the input will be rejected.

After successfully identifying the "from" and "to" positions, the structure is erased from the screen and redrawn with the "from" position now located at the "to" position on the screen. A message is then displayed to inform the user that the operation has been completed.

It should be noted that no changes are made to the ASSM as a result of this operation. The user may now add to or modify the structure using any of the previously described menu operations.

5.2.3.2 Zooming-Out on a Structure

This menu operation may be used on any structure which is currently displayed on the screen in its zoomed-in mode. The resulting display presents the entire structure on the screen, regardless of its size, in a color-coded form. Following is a legend for the nodes and their color-coded representations on a zoomed-out structure display:

ALPHA	- green rectangle
SUBNET	- green ellipse
EVENT	- yellow circle
VALIDATION_POINT	- blue circle
AND	- magenta circle
OR	- white circle
RETURN	- yellow square
ENTRY	- white square
TERMINATE	- white square
FOR EACH	- turquoise circle
INPUT_INTERFACE	- red circle
OUTPUT_INTERFACE	- red circle
SELECT	- red circle
Other	- blue rectangle

Once a structure has been displayed in its zoomed-out mode, none of the node manipulation menu operations are allowed; should the user attempt one of these operations, an error message will result and the menu selection will be ignored.

5.2.3.3 Zooming-In on a Structure

This operation can be used only if a structure is currently displayed in its zoomed-out mode (see the preceding section); otherwise, its selection will result in an error message and the selection will be ignored.

If the selection is allowed, the user responds by selecting a point on the zoomed-out structure. This point in the structure will be re-displayed at the top center of the display area. Below this will be displayed, in the zoomed-in mode, the remaining portion of the structure below the selected point. The user may then continue to add to or modify the structure as desired.

5.2.3.4 Displaying a Node Element

As indicated previously, only the first three characters of the name of an element associated with a node are displayed on the structure. By selecting the DISPLAY NODE menu operation and subsequently pointing to the desired node via the trackball, the user is able to see the entire element name. It will be displayed just below the color menu line. If the selected node has no element associated with it, such as an AND node, an appropriate message will result and the node selection is ignored.

5.2.3.5 Displaying a Branch

Once the conditional expression on an OR/FOR EACH node branch has been entered in the ASSM, it is no longer visible on the displayed structure. When modifying a branch extending from an OR/FOR EACH node, it may be necessary to know the associated conditional expression. By selecting the DISPLAY BRANCH menu operation, the user may at any time have the conditional expression of any branch displayed. Following the menu selection, the user responds with trackball selections of the successor and predecessor nodes of the desired branch on the displayed structure. If no conditional expression exists for the selected branch, an appropriate message is displayed and the node selections are ignored. If the branch has an ordinal associated with it, the ordinal value will be displayed; the user will then be required to respond with a trackball entry in order to continue. The first line of the conditional expression will then be displayed. A trackball entry will again be required for display of each subsequent line of the conditional expression. After all lines of the conditional expression have been displayed, an appropriate message is displayed to inform the user that the operation is complete.

5.2.3.6 Displaying a Structure on CALCOMP

Using the CALCOMP menu operation, the user can have the currently displayed structure plotted on the CALCOMP plotter. Upon selecting this operation, the message "DO YOU WANT STANDARD DOCUMENT SIZE, KEYIN YES OR NO" will be displayed. The user responds by keying in YES (Y) or NO (N) via the keyboard. If the response is YES, the generated CALCOMP plot will be 8-1/2 x 11 inches. If the response is NO, the user will then be asked to keyin the desired document height and width via the keyboard. These values may be keyed-in as either integers or real numbers. A maximum of 29 inches for the document height and 50 inches for the width are allowed. Any values larger than those will be rejected and replaced by their respective maximums.

The nodes on the plotted structure will retain the same relative position on the structure as they appear on the screen. However, up to thirty characters of the element names associated with each node will be displayed on the plotted structure. A legend page will also be generated for each structure which has branches with conditional expressions and/or nodes with comments. These branches and nodes are numbered on the structure; these numbers, with their corresponding conditional expressions and/or comment, are displayed on the legend page (see Figure 5-3).

When the plotting is complete, an appropriate message is displayed to inform the user that other processing may be continued.

5.2.3.7 Displaying a Structure in the Prompting Mode

Structures residing in the ASSM may or may not contain graphics coordinate data for display on the ANAGRAPH console. If a structure without graphics coordinates is retrieved, the user is given the option to either allow the required coordinate data to be generated automatically or to use the prompting capability provided by the SUCCESSOR NODE menu operation. If the user selects the prompting capability, the entry node for the structure is immediately displayed at the top center of the screen display area and the user is advised to use the SUCCESSOR NODE menu operation for displaying subsequent nodes.

Upon selection of the SUCCESSOR NODE operation, the user responds by pointing to the node on the structure for which the successor node has not yet been displayed. If the identified node has no successor nodes to be displayed, an appropriate message results and the node selection is ignored. If the identified node has more than one successor node, all of which are not displayed, the user will be informed as to the number of successors and will be required to depress the trackball entry key to continue. The node type of the first successor node, or the successor node if there is only one successor, is displayed and the user is then required to use the trackball to select a point on the screen at which the successor node is to be displayed. If the selected point is too close to an existing node on the screen and might, therefore, cause a node overlap, then an appropriate message results and the inputs are rejected. Also, if the selected point is outside the screen display area or too close to the border such that the node, if displayed, would extend outside the screen display area, an appropriate message is displayed and the inputs are rejected. In either case, the user must re-issue the node selection in order to continue. The above process is repeated until all nodes on the structure have been displayed.

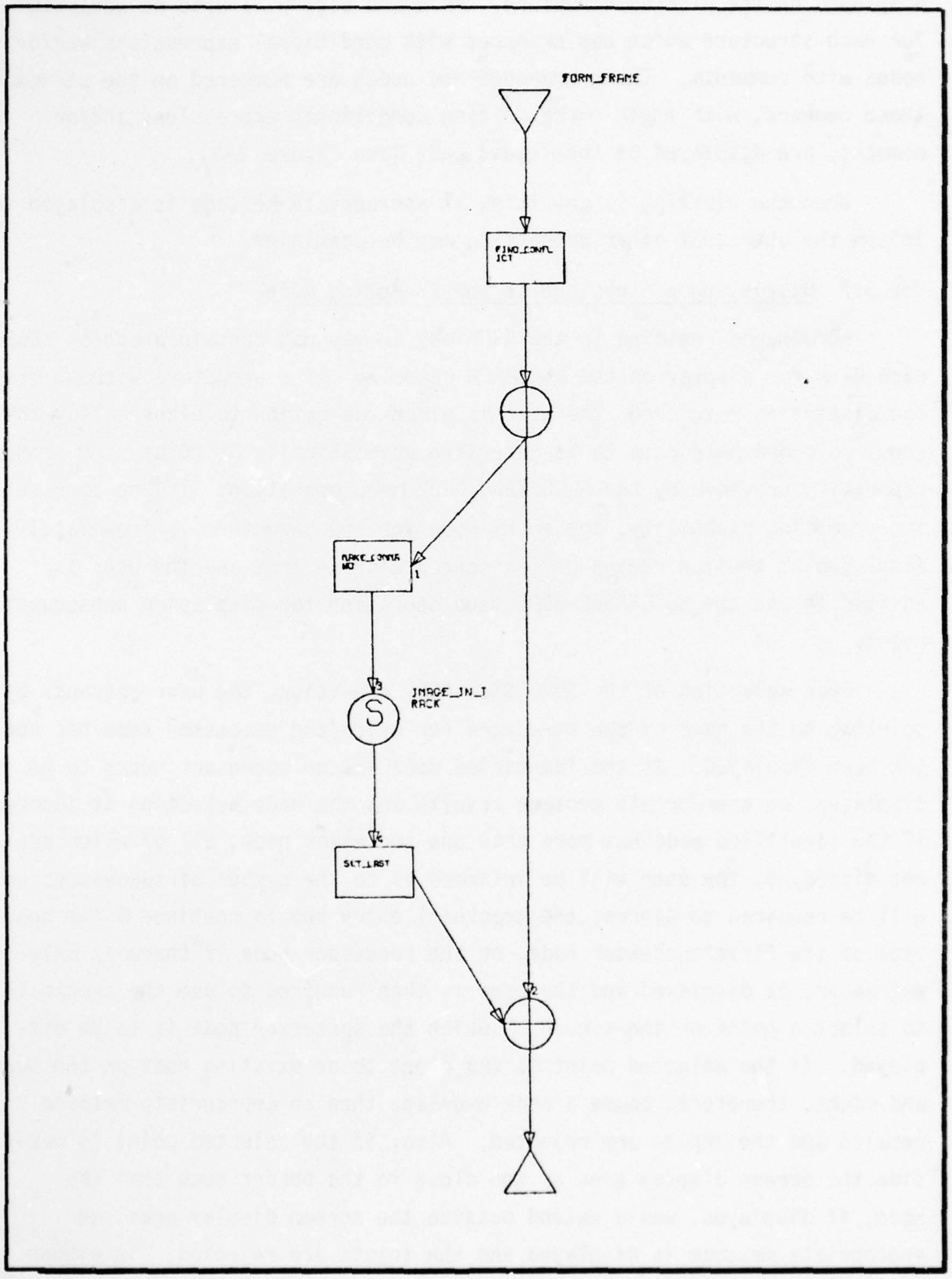
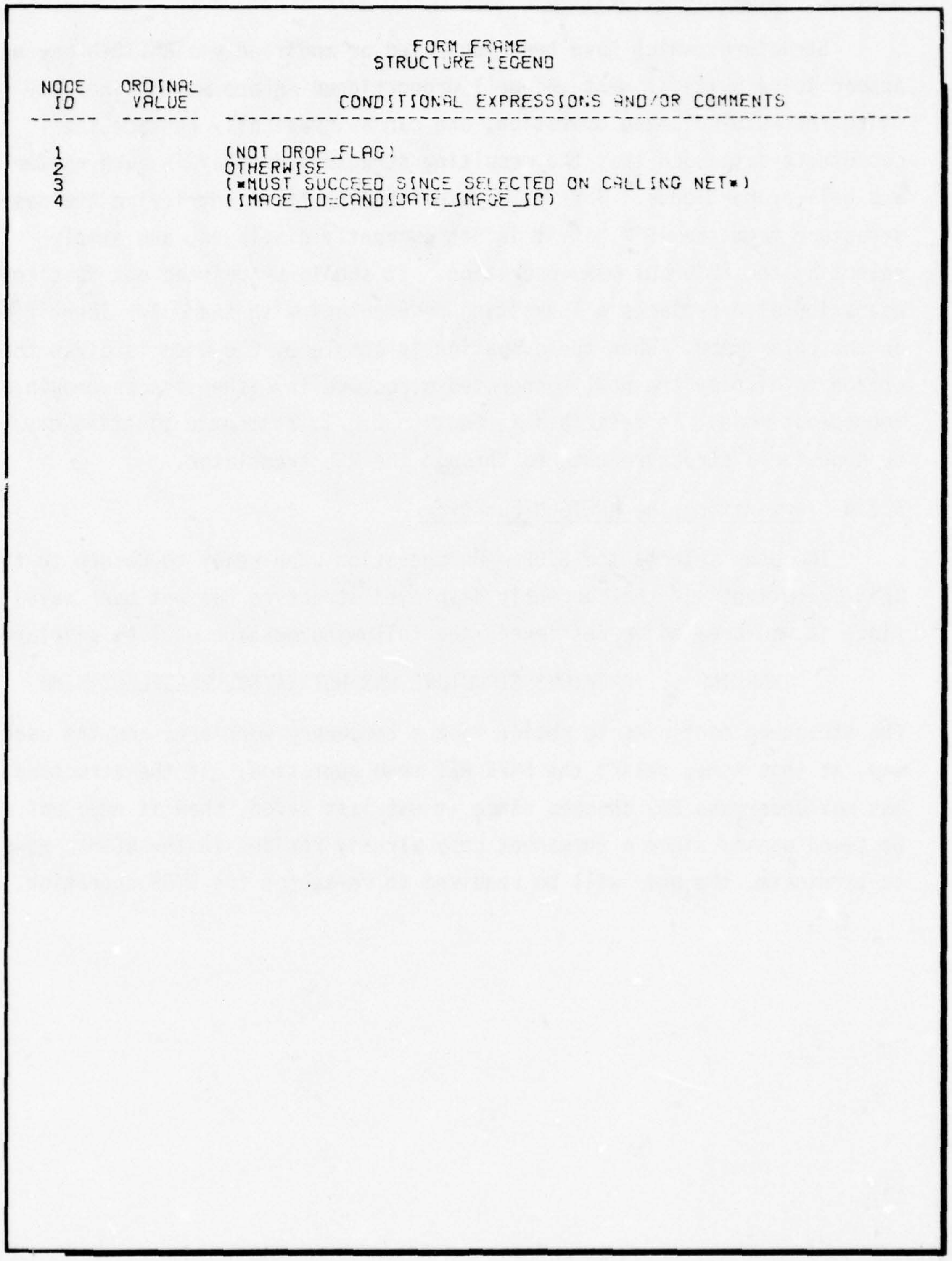


Figure 5-3 CALCOMP Display

BEST AVAILABLE COPY



```
                                   FORM_FRAME
                                STRUCTURE LEGEND
 NODE    ORDINAL
  ID      VALUE                  CONDITIONAL EXPRESSIONS AND/OR COMMENTS
------   -------   --------------------------------------------------------------

  1                (NOT DROP_FLAG)
  2                OTHERWISE
  3                (*MUST SUCCEED SINCE SELECTED ON CALLING NET*)
  4                (IMAGE_ID=CANDIDATE_IMAGE_ID)
```

Figure 5-3 CALCOMP Display (Continued)

BEST AVAILABLE COPY

5.2.3.8 Automatic Displaying

Structures which have been generated or modified via RNETGEN may not appear to be quite as neat and well proportioned as one would like. By using the AUTOPLOT menu operation, one can automatically replace the coordinate data such that the resulting structure display is much neater and well proportioned. This is accomplished by first retrieving the desired structure from the ASSM, if it is not currently displayed, and simply selecting the AUTOPLOT menu operation. It should be pointed out that this operation also replaces all existing node colors with the color identified in the color menu. When the operation is complete, the user is given the option to display the newly generated structure in either its zoomed-in or zoomed-out mode. As described in Section 5.2.1, automatic plotting may be done for a structure entered through the RSL translator.

5.2.4 Terminating the RNETGEN Function

The user selects the STOP menu operation when ready to return to the REVS Executive. If the currently displayed structure has not been saved since it was created or retrieved, the following message will be displayed:

WARNING ... PREVIOUS STRUCTURE WAS NOT SAVED, RE-SELECT MENU

The structure continues to reside in the temporary work area and the user may, at this time, select the SAVE NET menu operation. If the structure has not undergone any changes since it was last saved, then it need not be saved again, since a permanent copy already resides in the ASSM. However, to terminate, the user will be required to re-select the STOP operation.

6.0 ANALYZING AND DISPLAYING REQUIREMENTS (RADX FUNCTION)

The use of the Requirements Analysis and Data Extraction (RADX) function is described in this section. The three major areas of this generalized tool are explained in detail in the following sections. The define set facility which provides an ASSM query and interrogation capability and provides a technique for identifying subsets of information to be processed by other RADX commands is described in Section 6.1. The various ways to extract information from the ASSM and generate printed, punched, or plotted output is presented in Section 6.2. A description of the automated static analysis performed by RADX is contained in Section 6.3.

General information that applies to all commands that are input to the RADX function and an explanation of RADX output messages are provided below.

RADX Input Statements

As each RADX command is described in subsequent sections, the syntax will be presented along with an explanation of the meaning of the command. The command syntax presented is expressed in the extended Backus-Naur Form (BNF) explained in Appendix A. The complete syntax of the RADX command statements is summarized in Appendix F.1.

There are some syntax and semantic rules which are not easily defined in syntax diagrams or BNF notation and do not warrant repetition as each command is described; these rules are summarized below.

- A RADX statement is terminated by a period followed by a space that is not contained in a comment or text string.
- All RADX names (e.g., set name) must be 60 or less characters in length. The first character can be an underscore or a letter. The remaining characters can be an underscore, a letter, or a digit.
- All numbers are in standard PASCAL form (See Appendix B).
- A comment, a sequence of characters beginning with (* and ending with *), may be placed at any location in a statement. For example, (* THIS IS A COMMENT *).
- A text string, a sequence of characters beginning and ending with double quotes, may be used only where specified in the RADX command syntax.

- The characters comma, colon, and semicolon are optional punctuation marks that can be used any place in a statement without changing the meaning of the statement.
- Special processing is given to the SYNONYM element type. If a SYNONYM element is named in a RADX command and it EQUATES to another element, then the other element is used in place of the SYNONYM element. For the case that the SYNONYM is not EQUATED, the SYNONYM element is used.
- An element name cannot be preceded by an element type name to identify a single element. When an element type name is used in a command, it is interpreted as a reference to all the elements of the element type.

Display of User's Command

A RADX command that is input by the user and the display that results from the command are separated by preceding the command with the header

```
[RADX COMMAND =
```

and following the command with a line of underscores that is as long as the longest line in the command. When RADX is being executed on-line, the header also serves as a prompter to indicate that an input from the user is expected.

Error Messages

A summary of the error messages that can be issued by RADX is given in Appendix F.2. All error messages have a standard first line format that has the following general form:

```
*ERROR XXXX description.
```

The XXXX is a unique four digit number that relates to the area of RADX that detected the error and description is an explanation of the error.

Error messages issued by RADX can occur during the translation of a statement or during the processing of a statement after it is successfully translated. For errors that occur during translation, an output line is displayed after the error message to identify the cause of the error. The line will contain the keyword SYMBOL followed by the user's input symbol that caused the error. Error messages that occur during the processing of a statement have supplementary information following the message when it is needed to further identify the source of the error.

A summary of the number of error messages issued by RADX during a single execution of the function is also posted in the REVS.LOG file. This appears as:

```
QQ 001 NUMBER OF ERROR MESSAGES ISSUED BY RADX = XX.
```

6.1 SUBSETTING A REQUIREMENTS DATA BASE

The define set facility of RADX provides a mechanism to subset an ASSM for the purpose of user-directed analysis of the data base and user selection of requirements information to be subsequently processed by other RADX commands such as LIST and PLOT.

Conceptually, a RADX set is a named collection of elements in the ASSM. There are two basic types of sets that can be used: predefined sets and user defined sets.

Predefined Sets

The predefined sets provide a basic subsetting of the ASSM. They are always defined and available for reference when RADX is activated and cannot be redefined by the user. The predefined sets are:

- The universal set, which is referred to as ALL or ANY, contains all the elements in the ASSM.
- Element type sets, which are referred to by an element type name, contain all the elements that are of the named element type.
- Element sets, which are referred to by an element name, contain the named element.

To illustrate the predefined sets, assume that an ASSM contains only the following elements:

```
ALPHA: A.
DATA:  X.
DATA:  Y.
```

When RADX is executed the following sets would immediately be ready for use:

```
SET ALL = {A, X, Y}        SET A = {A}
SET ANY = {A, X, Y}        SET X = {X}
SET ALPHA = {A}            SET Y = {Y}.
SET DATA = {X, Y}
```

User Defined Sets

Additional subsetting of the ASSM can be achieved by defining new sets by one of the methods described in this section. These new sets are

PRECEDING PAGE NOT FILMED
BLANK

referred to as user defined sets and they are specified using the following general syntax:

```
SET name = description.
```

The name given to the set can be any word that is not contained in the ASSM and that is not one of the following keywords: ALL, ANY, SET, ELEMENT_TYPE, MULTIPLE, or RSL. Description specifies the condition for membership in the new set.

After a set is defined, a display is made by RADX that tells the number of members that are contained in the new set. This will appear in the output as:

```
SET COUNT = XX.
```

Referencing Sets

A RADX set is referenced in a command by the following **<set identifier>** syntax:

```
[SET] name
```

The name can be that of a predefined set or a user defined set.

The syntax of the various set description methods appearing in the remainder of this section use the symbol <set identifier> to designate a set identifier. In the explanations, different terms for the set identifier are used to indicate the meaning of the command. The terms used are: existing set, first independent set, second independent set, subject set, object set, **and candidate set.**

The following are examples of set identifiers that can be used in RADX commands to reference sets:

```
ALL
SET ALL
DATA
SET DATA
X
SET X.
```

Redefining Sets

As previously mentioned, a predefined set cannot be redefined. However, user defined sets can be redefined at a user's convenience. It is permissible to use a set as an independent and dependent set in the same command. For example:

SET X = SET X OR SET Y.

In the command, the contents of the independent set X (i.e., the one on the right side of the equal sign) remains constant during the processing of the command. When the command is completed, the old contents of the set are deleted and the new definition takes effect.

The following is a list of the different ways that a set can be defined by the user. They are described in the subsequent subsections.

- enumeration
- combination
- attribute qualification
- relationship qualification
- structure qualification
- hierarchy qualification

This section concludes with examples of how the definition of sets can be employed to perform a user-directed analysis of a requirements data base.

6.1.1 Defining Sets By Enumeration

The members of a set can be defined as those contained in another set or as the union of a list of sets by the following syntax:.

$$\text{SET new-set-name} = \left\{ \text{<set identifier>} \right\}_1^n.$$

The set identifier designates either a predefined set or a set defined by the user in a previous RADX statement.

The statements given below demonstrate the use of this technique for defining sets.

```
SET A = ALPHA, FILE, INPUT_INTERFACE.

SET B = SET A, STATE, FOUND.
```

In the first statement, set A will contain all the elements in the ASSM that are members of the predefined element type sets ALPHA, FILE or INPUT_INTERFACE. The set B will contain elements that are members of the user defined set A plus the two predefined element sets STATE and FOUND.

6.1.2 Defining Sets By Combination

A set can be defined as the logical combination of two existing independent sets by a statement using the following syntax:

SET new-set-name = <set identifier> $\left\{\begin{matrix}\text{AND}\\\text{OR}\\\text{MINUS}\end{matrix}\right\}_1^1$ <set identifier>.

The set identifiers designate a first independent set and a second independent set, respectively. The type of combination performed is indicated by the operation connecting the two independent sets. These operations are:

AND	- Set intersection. The members of the new set are those that are members of both the first independent set and the second independent set.
OR	- Set union. The members of the new set are those that are members of either the first independent set or the second independent set.
MINUS	- Set difference. The members of the new set are those that are members of the first independent set but not the second independent set.

The following are examples of set definitions by combination.

```
SET ALPHA_DATA = ALPHA OR DATA.

SET NETS_IN_X = R_NET AND SET X.

SET ALL_EXCEPT_DECISION = ALL MINUS DECISION.
```

6.1.3 Defining Sets By Attribute Qualification

The qualify by attribute statement provides the capability to define a set of elements that have or do not have certain attribute characteristics. The syntax of the statement is:

SET new-set-name = <set identifier> $\left\{\begin{matrix}\text{<positive connector>}\\\text{<negative connector>}\end{matrix}\right\}_1^1$

<attribute criterion>.

The attribute criterion can be specified by one of the following forms:

1) attribute-name

2) attribute-name [<relation operator>] <value>.

The members of the new set are those members of the subject set (designated by the set identifier) that satisfy the attribute criterion in the manner indicated by the connector. When a positive connector is used, the members of the new set are those in the subject set which have the characteristics stated by the attribute criterion. If a negative connector is used, then the members of the new set are those in the subject set that do not have the attribute criterion.

A variety of terms are allowed to be used as positive and negative connectors to increase the readability of RADX statements. The allowable terms are listed below:

<u>positive connectors</u>	<u>negative connectors</u>
WITH	*<positive connector>* NO
WHERE	<positive connector> NOT
WHICH	WITHOUT
WHICH IS	
IN	
FROM	
SUCH	
SUCH THAT	
THAT	
THAT IS	
BY	

<u>Qualify By Attribute Instance</u>

When the attribute criterion is specified as attribute name, the condition for membership in the new set is independent of attribute values. It depends only on an instance of the attribute being present in the ASSM when a positive connector is used and absent from the ASSM when a negative connector is used. The next statements are valid examples of this type of set definition.

```
SET INT_VAL = DATA WITH INITIAL_VALUE.

SET NO_LOCALITY = ALL WITHOUT LOCALITY.
```

Qualify By Attribute Value

The second form of the attribute criterion,(i.e., attribute-name [<relational operator>] value) allows the definition of the new set to be based on attribute values. When a positive connector is used with this form, the new set will contain members of the subject set that have an attribute value which satisfies the relational operator. The use of a negative connector will cause the new set to contain members of the subject set that do not satisfy the relational operator. If the optional relational operator is not specified, the equal sign (i.e., test for equality) is assumed. A list of legal relational operators is provided in the following table.

Relation Operator	Meaning
>	Greater than
>=	Greater than or equal
<>	Not equal
=	Equal
<=	Less than or equal
<	Less than

The value that is specified in the attribute criterion can be an integer or real number, a value name, or a text string that is not longer than 60 characters. The relational operators = and <> are the only ones that are legal if the value is specified as a text string or as a value name.

The next group of statements are examples of the definition of sets based on attribute values.

```
SET INT_VAL_ZERO = DATA WITH INITIAL_VALUE 0.

SET INT_VAL_GT_ZERO = DATA WITH INITIAL_VALUE > 0.

SET GAMMA_DATA = DATA WITH USE = GAMMA.

SET MY_ALPHAS = ALPHA SUCH THAT ENTERED_BY = "THIS IS MINE".
```

6.1.4 Defining Sets By Relationship Qualification

A set can be defined to contain elements with or without certain relationship characteristics with the qualify by relationship statement. The syntax of the statement is:

SET new-set-name = <set identifier> $\left\{\begin{matrix}\text{<positive connector>}\\ \text{<negative connector>}\end{matrix}\right\}_1^1$

[MULTIPLE] relation-name $\left\{\text{relation-optional-word}\right\}_0^n$

[<set identifier>].

The first set identifier selects a subject set of candidate members that can be included in the new set. The legal terms that can be used for positive connector and negative connector are described in Section 6.1.3. The MULTIPLE option is used to indicate that an element in the subject set must have more than one instance of the specified relationship before it can be included in the new set. The relation name can be a primary relation or a complementary relation. Any number of RSL relational optional words may appear after the relationship to increase the readability of the statement. The optional object set (identified by the second set identifier) is used to indicate that an element in the subject set must have a relationship instance with one or more of the elements in the object set to satisfy the qualification criterion. If the object set is not specified, an element in the subject set is qualified for inclusion in the new set by having an instance of the relationship with any element in the ASSM.

The following examples demonstrate uses of the qualify by relationship statement. The RADX comment that appears with a statement describes the contents of the set being defined.

```
SET USED_DATA = DATA THAT IS INPUT
    (*DATA ELEMENTS WHICH HAVE AN INSTANCE OF THE INPUT
      RELATIONSHIP.*).

SET NOT_USED_DATA = DATA THAT IS NOT INPUT
    (*DATA ELEMENTS WITHOUT AN INSTANCE OF THE INPUT
      RELATIONSHIP.*).

SET USED_BY_ALPHA_UPDATE_STATE = DATA THAT IS INPUT TO UPDATE_STATE
    (*DATA ELEMENTS THAT ARE INPUT TO UPDATE_STATE.*).

SET ERROR_1 = DATA THAT IS MULTIPLE CONTAINED
    (*DATA ELEMENTS WITH MORE THAN ONE INSTANCE OF THE
      CONTAINED RELATIONSHIP.*).

SET SIMPLE = MESSAGE THAT IS NOT MULTIPLE MADE BY FILE
    (*MESSAGE ELEMENTS THAT ARE NOT MADE BY MORE THAN ONE FILE.*).
```

6.1.5 Defining Sets By Structure Qualification

Implicit relationships between structures and elements used in structures may be used for defining a new set of elements that have or do not have certain structural characteristics. These implicit relationships are named REFERS and REFERRED. They cannot be explicitly input through the RSL translator but they are implicitly defined when a structure is entered in the ASSM.

The REFERS relationship exists between an element with a structure and the elements used on the structure. The REFERRED relationship is the complement of the REFERS relationship. These implicit relationships are used in the same manner as RSL relationships are used to define a set by relationship qualification as explained in Section 6.1.4. The syntax for using REFERS and REFERRED is:

$$\text{SET new-set-name = <set identifier>} \left\{\begin{matrix}\text{<positive connector>}\\ \text{<negative connector>}\end{matrix}\right\}_1^1$$

$$\text{[MULTIPLE]} \left\{\begin{matrix}\text{REFERS}\\ \text{REFERRED}\end{matrix}\right\}_1^1 \left\{\text{relational-optional-word}\right\}_0^n$$

[<set identifier>].

The following examples illustrate different uses of this statement.

```
SET R_NET_NO_STRUCTURE = R_NET WITHOUT REFERS.

SET R_NET_USING_UPDATE_STATE = R_NET WHICH REFERS TO UPDATE_STATE.

SET ALPHAS_NOT_USED = ALPHA THAT IS NOT REFERRED.

SET ALL_NEEDED_BY_R_NET_RADAR_SUMMARY = ALL THAT IS REFERRED TO
    BY RADAR_SUMMARY.
```

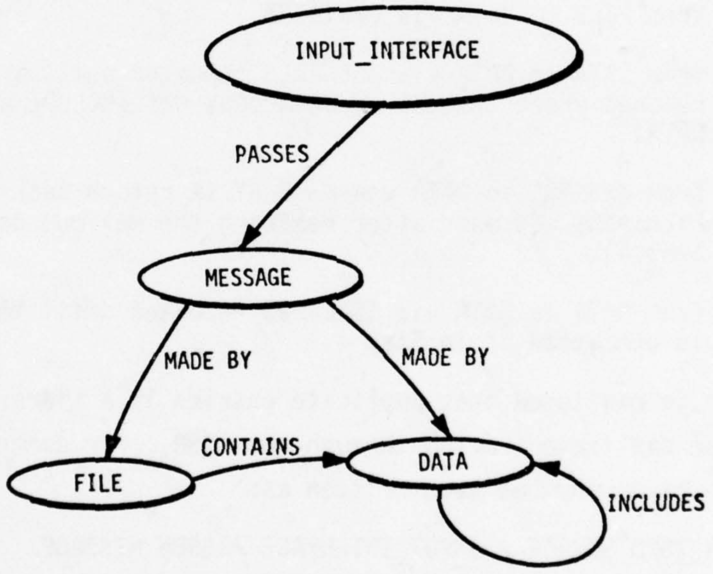
6.1.6 Defining Hierarchies

There are several hierarchies that exist in the definition of RSL such as data hierarchies and structure hierarchies that can be identified to RADX and used later as a "road map" to trace through the ASSM for the purpose of defining a set or determining the order to extract and display information. A RADX hierarchy is defined using the following syntax:

$$\left\{\begin{matrix}\text{HIERARCHY}\\ \text{HIER}\end{matrix}\right\}_1^1 \text{ hierarchy-name = } \left\{ \text{<set identifier> <binding relation>} \left\{\text{optional-word}\right\}_0^n \text{ <set identifier>}\right\}_1^n .$$

In the statement, hierarchy name is a unique name that will be used to reference the hierarchy, the set identifiers designate sets that must be defined before the hierarchy is defined, and binding relation is any RSL relation or an implicit relation, REFERS or REFERRED.

For example, the following graph illustrates an RSL information hierarchy that can exist in the requirements data base:



The nodes in the graph represent sets (in this case predefined element type sets) and the branches represent binding relationships between the sets. This hierarchy can be named, say INFO_SOURCE, and input to RADX for future use by defining the connectivity of the graph (i.e., hierarchy) as follows:

```
HIER INFO_SOURCE = INPUT_INTERFACE PASSES MESSAGE;
                   MESSAGE MADE BY FILE;
                   MESSAGE MADE BY DATA;
                   FILE CONTAINS DATA;
                   DATA INCLUDES DATA.
```

When this command is input to RADX, it is translated and stored for future reference by other RADX commands. In addition to the connectivity information between the hierarchy entries, a top-of-the-hierarchy is implicitly defined as the first set that appears in the hierarchy description. For the above example INPUT_INTERFACE is the top-of-the-hierarchy. This will be used to determine the starting point in the "road map" when a trace through the ASSM is made following the hierarchy.

Whenever a trace is performed, it is done in a depth-first manner. For the example, the order of tracing would occur as follows:

1) from INPUT_INTERFACE to MESSAGE via PASSES.

2) from MESSAGE to FILE via MADE BY.

3) from FILE to DATA via CONTAINS.

4) from DATA to DATA via INCLUDES repeated until a point is reached where the set of DATA does not INCLUDE any other DATA.

5) from MESSAGE to DATA via MADE BY (a return back up the hierarchy was made after reaching the maximum depth in Step 4).

6) from DATA to DATA via INCLUDES repeated until the trace is exhausted as in Step 4.

The user is cautioned that duplicate entries in a hierarchy definition will cause duplicate tracing through the ASSM. For example, suppose that the above hierarchy had been written as:

```
HIER INFO_SOURCE = INPUT_INTERFACE PASSES MESSAGE;
                   MESSAGE MADE BY FILE;
                   FILE CONTAINS DATA;
                   DATA INCLUDES DATA;
                   MESSAGE MADE BY DATA;
                   DATA INCLUDES DATA.
```

Since there are two entries for DATA INCLUDES DATA, lines 4 and 6, many duplicate traces would occur. For example, after entry 3 traces from FILE to DATA via CONTAINS, entry 4 would trace from DATA to DATA via INCLUDES and later entry 6 would be processed causing a duplication of what was done at entry 4, tracing from DATA to DATA via INCLUDES. The same type of activity would occur following the processing of entry 5 which is a trace from MESSAGE to DATA via MADE BY.

6.1.7 Defining Sets By Hierarchy Qualification

This capability is used to define a set of elements as those which are members of a given set and are reached while tracing a **hierarchy** as defined in Section 6.1.6. The syntax of this define set statement is:

$$\text{SET new-set-name = <set identifier> <positive connector>} \left\{\begin{matrix}\text{HIERARCHY}\\\text{HIER}\end{matrix}\right\}_1^1 \text{hierarchy-name.}$$

The set identifier selects a set which contains candidate members for inclusion in the new set, the positive connector is one of the terms defined in Section 6.1.3, and hierarchy name is the name of a previously defined hierarchy.

For an element to become a member of the new set, it must satisfy the following criteria:

1) it must be a member of the candidate set

2) it must be encountered while traversing the hierarchy.

Some examples of how a hierarchy is used to define a set are given next using the following hierarchy definition.

```
SET SOURCE_INTERFACE = INPUT_INTERFACE.

HIERARCHY INFO_SOURCE = SOURCE_INTERFACE PASSES MESSAGE;
                        MESSAGE MADE BY FILE;
                        MESSAGE MADE BY DATA;
                        FILE CONTAINS DATA;
                        DATA INCLUDES DATA.
```

Notice that in this example the user defined SET: SOURCE_INTERFACE is the top-of-the-hierarchy. It is required that the set be defined before the hierarchy is defined but the set can be redefined as will be shown in one of the examples that follows.

The following examples will describe the desired contents of a set and the RADX commands that can be used to accomplish the objective.

Example 1

OBJECTIVE: A SET X which contains all the INPUT_INTERFACEs in the ASSM, the MESSAGEs that PASS them, all the FILEs and DATA that MAKE the MESSAGEs, and all the DATA that is CONTAINED in the FILEs.

COMMAND:

```
SET X = ALL IN HIER INFO_SOURCE.
```

Example 2

OBJECTIVE: A SET Y consisting of the INPUT_INTERFACE: RADAR_IN, the MESSAGEs that PASS RADAR_IN, the FILEs and DATA that MAKE the MESSAGEs, and the DATA that is CONTAINED in the FILEs.

COMMANDS:

```
SET SOURCE_INTERFACE = RADAR_IN.
SET Y = ALL IN HIER INFO_SOURCE.
```

Example 3

OBJECTIVE: Same as Example 2.

COMMANDS:

```
SET SOURCE_INTERFACE = INPUT_INTERFACE.
SET RADAR_TRACE = RADAR_IN, MESSAGE, FILE, DATA.
SET Y = RADAR_TRACE FROM HIERARCHY INFO_SOURCE.
```

Example 4

OBJECTIVE: A SET Z that contains only the DATA which is a part of the INPUT_INTERFACE because it MAKES a MESSAGE or is CONTAINED in a FILE that MAKES a MESSAGE which PASSES the INPUT_INTERFACE.

COMMANDS:

```
SET SOURCE_INTERFACE = INPUT_INTERFACE.
SET SOURCE_TRACE = INPUT_INTERFACE OR DATA.
SET TEMP = SOURCE_TRACE IN HIER INFO_SOURCE.
SET Z = SET TEMP MINUS INPUT_INTERFACE.
```

NOTE: When SET TEMP was defined it contained only element types INPUT_INTERFACE and DATA. The set did not contain any MESSAGEs or FILEs because they were not in SET: SOURCE_TRACE which was the original set being qualified. With the removal of INPUT_INTERFACEs from SET TEMP, the objective is satisfied for SET Z. The reason that INPUT_INTERFACE needs to be in SOURCE_TRACE is to satisfy the requirement that the candidate set being qualified must contain the starting points for the trace.

6.1.8 Using Sets to Analyze a Requirements Data Base

One of the primary purposes for the set definition facility of RADX is to provide a technique that allows a user to analyze a requirements data base. Table 6.1 contains RADX commands that can be used to analyze a data base for compliance with the RSL conventions described in Section 3.0. The commands are grouped according to the area of the requirements specifications that they analyze. Those sets which have a comment identify a violation of a convention if they are not empty. The comment describes the violated convention. The sets without a comment are used in a temporary manner for building the sets that identify errors. The table also includes the hierarchy definitions that are required to form the sets.

The LIST command that is described in Section 6.2 can be used to display the contents of the sets which identify errors in order to document the anomalies that are present in an ASSM.

Table 6.1 Examples of RADX Commands For Requirements Analysis

COMMANDS TO TEST SUBSYSTEM AND INTERFACE SPECIFICATIONS

```
SET UNCONNECTED_SUBSYSTEM = SUBSYSTEM THAT IS NOT CONNECTED
                     (* ALL SUBSYSTEMS MUST BE CONNECTED. *).
SET INTERFACE = INPUT_INTERFACE OR OUTPUT_INTERFACE.
SET INTERFACE_NOT_CONNECTED = INTERFACE WITHOUT CONNECTS
                     (* AN INTERFACE MUST CONNECT TO A SUBSYSTEM. *).
SET TOO_MANY_CONNECTS = INTERFACE THAT MULTIPLE CONNECTS
                     (* AN INTERFACE CANNOT CONNECT TO MORE THAN
                        ONE SUBSYSTEM. *).
SET INTERFACE_NO_MESSAGE = INTERFACE WITHOUT PASSES
                     (* AN INTERFACE MUST PASS AT LEAST ONE MESSAGE. *).
SET OUT_MSG = MESSAGE THAT PASSED OUTPUT_INTERFACE.
SET OUT_MSG_NOT_FORMED = OUT_MSG THAT IS NOT FORMED
                     (* ALL MESSAGES THAT PASS AN OUTPUT_INTERFACE
                        MUST BE FORMED. *).
SET MSG_NOT_PASSED = MESSAGE THAT IS NOT PASSED
                     (* A MESSAGE MUST BE PASSED BY EITHER AN INPUT
                        OR OUTPUT INTERFACE. *).
SET MULTI_PASSED_MESSAGE = MESSAGE THAT IS MULTIPLE PASSED
                     (* A MESSAGE CAN ONLY PASS ONE INTERFACE. *).
SET MULTI_USED_INPUT_INF = INPUT_INTERFACE THAT IS MULTIPLE REFERRED
                     (* AN INPUT_INTERFACE CANNOT BE REFERENCED
                        BY MORE THAN ONE R_NET. *).
```

COMMANDS TO TEST STRUCTURE SPECIFICATIONS

```
SET UNENABLED_R_NETS = R_NET THAT IS NOT ENABLED
                     (* R_NETS MUST BE ENABLED. *).
SET REF_INPUT_INF = R_NET THAT REFERS TO INPUT_INTERFACE.
SET MISSING_INF_ENABLE = REF_INPUT_INF THAT IS NOT ENABLED
                           BY INPUT_INTERFACE
                     (* AN R_NET THAT REFERENCES AN INPUT_INTERFACE
                        MUST BE ENABLED BY THE INTERFACE. *).
SET BAD_MULTI_ENABLE = REF_INPUT_INF THAT IS MULTIPLE ENABLED
                     (* AN R_NET WHICH REFERENCES AN
                        INPUT_INTERFACE CAN ONLY BE ENABLED
                        BY THE INTERFACE. *).
SET NOT_REF_INPUT_INF = R_NET MINUS REF_INPUT_INF.
SET BAD_INTERFACE_ENABLEMENT = NOT_REF_INPUT_INF THAT IS
                     ENABLED BY INPUT_INTERFACE
                     (* AN R_NET SHOULD NOT BE ENABLED BY AN
                        INPUT_INTERFACE UNLESS THE INTERFACE
                        APPERAS IN THE R_NET STRUCTURE. *).
SET STRUCTURE_NODES = ALPHA, SUBNET, EVENT, VALIDATION_POINT,
                    INPUT_INTERFACE, OUTPUT_INTERFACE.
SET NETS = R_NET OR SUBNET.
SET UNUSED_NODES = STRUCTURE_NODES SUCH THAT NOT REFERRED TO BY NETS
                     (* FOR THE REQUIREMENTS TO BE COMPLETE, ALL
                        ALPHA, SUBNET, EVENT, VALIDATION_POINT,
                        INPUT_INTERFACE, AND OUTPUT_INTERFACE
                        ELEMENTS MUST BE USED IN EITHER AN R_NET
                        OR SUBNET STRUCTURE. *).
```

Table 6.1 Examples of RADX Commands For Requirements Analysis (Continued)

COMMANDS TO TEST STRUCTURE SPECIFICATIONS (CONTINUED)

```
SET STRUCTURE_ELEMENTS = R_NET, SUBNET, VALIDATION_PATH.
SET MISSING_STRUCTURE = STRUCTURE_ELEMENTS WITHOUT REFERS
                        (* THE REQUIREMENTS ARE NOT COMPLETE UNTIL
                           ALL R_NET, SUBNET, AND VALIDATION_PATH
                           ELEMENTS HAVE BEEN GIVEN A STRUCTURE. *).
SET NON_ENABLING_EVENT = EVENT WITHOUT ENABLES
                        (* AN EVENT MUST ENABLE AT LEAST ONE R_NET. *).
SET COMPLEX_DATA = DATA THAT INCLUDES DATA.
SET BAD_DELAYED_EVENT = EVENT THAT IS DELAYED BY COMPLEX_DATA
                        (* AN EVENT CAN ONLY BE DELAYED BY
                           LOWEST LEVEL DATA. *).
SET NON_ENABLING_INPUT_INF = INPUT_INTERFACE WITHOUT ENABLES
                        (* AN INPUT_INTERFACE MUST ENABLE AN R_NET. *).
SET MULTI_DELAYED = EVENT THAT IS MULTIPLE DELAYED
                        (* AN EVENT CANNOT BE DELAYED BY MORE THAN ONE
                           DATA ELEMENT. *).
```

COMMANDS TO TEST INFORMATION MEMBERSHIP SPECIFICATIONS

```
SET MULTI_CONTAINED = DATA THAT IS MULTIPLE CONTAINED
                        (* A DATA ITEM CAN ONLY BE CONTAINED
                           IN ONE FILE. *).
SET INFO = DATA OR FILE.
SET MULTI_ASSOCIATES_CLASS = INFO WHICH IS MULTIPLE ASSOCIATED
                                   WITH ENTITY_CLASS
                        (* A SINGLE DATA OR FILE CAN ONLY BE ASSOCIATED
                           WITH ONE ENTITY_CLASS. *).
HIER ENTITY_TYPE_TRACE = ENTITY_TYPE ASSOCIATES FILE,
                        ENTITY_TYPE ASSOCIATES DATA,
                        DATA INCLUDES DATA.
HIER ENTITY_CLASS_TRACE = ENTITY_CLASS ASSOCIATES FILE,
                        ENTITY_CLASS ASSOCIATES DATA,
                        DATA INCLUDES DATA.
HIER MESSAGE_TRACE = MESSAGE MADE BY FILE,
                        MESSAGE MADE BY DATA,
                        DATA INCLUDES DATA.
HIER FILE_TRACE = FILE CONTAINS DATA,
                        DATA INCLUDES DATA.
SET TYPE_INFO = ALL IN HIER ENTITY_TYPE_TRACE.
SET TYPE_INFO = TYPE_INFO MINUS ENTITY_TYPE.
SET CLASS_INFO = ALL IN HIER ENTITY_CLASS_TRACE.
SET CLASS_INFO = CLASS_INFO MINUS ENTITY_CLASS.
SET FILE_INFO = ALL IN HIER FILE_TRACE.
SET FILE_INFO = FILE_INFO MINUS FILE.
SET TYPE_AND_CLASS_INFO = TYPE_INFO AND CLASS_INFO
                        (* A DATA OR FILE CANNOT BE ASSOCIATED WITH BOTH
                           AN ENTITY_CLASS AND AN ENTITY_TYPE. *).
SET ENTITY_INFO = TYPE_INFO OR CLASS_INFO.
SET ENTITY_AND_FILE_INFO = ENTITY_INFO AND FILE_INFO
                        (* DATA CANNOT BE BOTH CONTAINED IN A FILE
                           AND ASSOCIATED WITH AN ENTITY. *).
```

Table 6.1 Examples of RADX Commands For Requirements Analysis (Continued)

COMMANDS TO TEST INFORMATION MEMBERSHIP SPECIFICATIONS (CONTINUED)

```
SET MSG_INFO = ALL IN HIER MESSAGE_TRACE.
SET MSG_INFO = MSG_INFO MINUS MESSAGE.
SET ENTITY_AND_MSG_INFO = ENTITY_INFO AND MSG_INFO
                         (* A SINGLE DATA OR FILE CANNOT BOTH MAKE
                            A MESSAGE AND BE ASSOCIATED WITH AN ENTITY. *).
SET MSG_FILE_INFO = MSG_INFO AND FILE_INFO
                         (* A DATA ITEM CANNOT BOTH MAKE A MESSAGE
                            AND BE CONTAINED IN A FILE. *).
SET CLASS_NO_TYPE = ENTITY_CLASS WITHOUT COMPOSED ENTITY_TYPE
                         (* AN ENTITY_CLASS MUST BE COMPOSED OF AT
                            LEAST ONE ENTITY_TYPE. *).
SET TYPE_NO_CLASS = ENTITY_TYPE WITH NO COMPOSES
                         (* AN ENTITY_TYPE MUST COMPOSE AN ENTITY_CLASS. *).
SET MULTI_COMPOSES = ENTITY_TYPE THAT MULTIPLE COMPOSES
                         (* AN ENTITY_TYPE CANNOT COMPOSE MORE THAN
                            ONE ENTITY_CLASS. *).
SET MULTI_ORDERED = FILE THAT IS MULTIPLE ORDERED
                         (* A FILE CANNOT BE ORDERED BY MORE THAN ONE
                            DATA ELEMENT. *).
SET ORDERING_DATA = DATA THAT ORDERS FILE.
SET NEEDS_TO_BE_IN_FILE = ORDERING_DATA THAT IS NOT CONTAINED
                         (* A DATA ELEMENT THAT ORDERS A FILE MUST
                            BE CONTAINED IN THE FILE. *).
SET BAD_ORDERING_DATA = ORDERING_DATA THAT INCLUDES
                         (* ONLY LOWEST LEVEL DATA CAN ORDER A FILE. *).
SET EMPTY_FILE = FILE WITHOUT CONTAINS
                         (* A FILE MUST CONTAIN AT LEAST ONE DATA ITEM. *).
SET EMPTY_MESSAGE = MESSAGE THAT IS NOT MADE
                         (* A MESSAGE MUST BE MADE BY EITHER DATA OR
                            FILE ELEMENTS. *).
SET EMPTY_ENTITY_TYPE = ENTITY_TYPE WITHOUT ASSOCIATES
                         (* AN ENTITY_TYPE MUST ASSOCIATE AT LEAST ONE
                            DATA OR FILE ELEMENT. *).
```

COMMANDS TO TEST INFORMATION USAGE AND ASSIGNMENT SPECIFICATIONS

```
HIER SUBSYS_TO_DATA = SUBSYSTEM CONNECTED INPUT_INTERFACE,
                      INPUT_INTERFACE PASSES MESSAGE,
                      MESSAGE MADE BY FILE,
                      MESSAGE MADE BY DATA,
                      FILE CONTAINS DATA,
                      DATA INCLUDES DATA.
HIER DATA_TO_SUBSYS = SUBSYSTEM CONNECTED OUTPUT_INTERFACE,
                      OUTPUT_INTERFACE PASSES MESSAGE.
                      MESSAGE MADE BY FILE,
                      MESSAGE MADE BY DATA,
                      FILE CONTAINS DATA,
                      DATA INCLUDES DATA.
HIER ALPHA_IN = ALPHA INPUTS FILE,
                      ALPHA INPUTS DATA,
                      FILE CONTAINS DATA
                      DATA INCLUDES DATA.
```

BEST AVAILABLE COPY

Table 6.1 Examples of RADX Commands For Requirements Analysis (Continued)

COMMANDS TO TEST INFORMATION USAGE AND ASSIGNMENT SPECIFICATIONS (CONTINUED)

```
HIER VP_IN = VALIDATION_POINT RECORDS FILE,
                     VALIDATION_POINT RECORDS DATA,
                     FILE CONTAINS DATA,
                     DATA INCLUDES DATA.
HIER ALPHA_OUT = ALPHA OUTPUTS FILE,
                     ALPHA OUTPUTS DATA,
                     FILE CONTAINS DATA,
                     DATA INCLUDES DATA.
SET STANDARD_DATA = FOUND, RECORD_FOUND, CLOCK_TIME.
SET SPECIFIED_DATA = DATA MINUS STANDARD_DATA.
SET SOURCE_1 = ALL IN HIER SUBSYS_TO_DATA.
SET SOURCE_2 = ALL IN HIER ALPHA_OUT.
SET SOURCE_3 = DATA WITH INITIAL_VALUE.
SET SOURCES = SOURCE_1, SOURCE_2, SOURCE_3.
SET SOURCES = SOURCES AND SPECIFIED_DATA.
SET NO_SOURCE = SPECIFIED_DATA MINUS SOURCES.
SET SINK_1 = ALL IN HIER DATA_TO_SUBSYS.
SET SINK_2 = ALL IN HIER ALPHA_IN.
SET SINK_3 = ALL IN HIER VP_IN.
SET SINK_4 = DATA THAT IS REFERRED.
SET SINK_5 = DATA WHICH DELAYS EVENT.
SET SINK_6 = DATA WHICH ORDERS FILE.
SET SINKS = SINK_1, SINK_2, SINK_3, SINK_4, SINK_5, SINK_6.
SET SINKS = SINKS AND SPECIFIED_DATA.
SET NO_SINK = SPECIFIED_DATA MINUS SINKS.
SET DATA_NO_SOURCE_AND_SINK = NO_SOURCE AND NO_SINK
                     (* A DATA ITEM SHOULD HAVE A SOURCE AND
                        A SINK. *).
SET DATA_WITH_SOURCE_BUT_NO_SINK = SOURCES MINUS SINKS
                     (* IF A DATA ITEM HAS A SINK, THEN IT MUST
                        HAVE A SINK. *).
SET DATA_WITH_SINK_BUT_NO_SOURCE = SINKS MINUS SOURCES
                     (* IF A DATA ITEM HAS A SINK, THEN IT MUST HAVE
                        A SOURCE. *).
```

COMMANDS TO TEST ENTITY OPERATION SPECIFICATIONS

```
SET CLASS_NOT_CREATED = ENTITY_CLASS THAT IS NOT CREATED
                     (* ALL ENTITY_CLASSES MUST BE CREATED. *).
SET CLASS_NOT_DESTROYED = ENTITY_CLASS THAT IS NOT DESTROYED
                     (* THE ANALYST SHOULD REVIEW THE REASON WHY
                        AN ENTITY_CLASS IS NOT DESTROYED. *).
SET TYPE_NOT_SET = ENTITY_TYPE THAT IS NOT SET
                     (* EVERY ENTITY_TYPE MUST BE SET. *).
SET TYPE_NOT_REFERENCED = ENTITY_TYPE THAT IS NOT REFERRED.
SET CLASS_OF_UNREFERRED = ENTITY_CLASS SUCH THAT COMPOSED
                           OF TYPE_NOT_REFERENCED.
SET CLASS_NOT_REFERRED = CLASS_OF_UNREFERRED THAT IS NOT REFERRED
                     (* EACH ENTITY_CLASS MUST BE DIRECTLY USED ON
                        A STRUCTURE OR INDIRECTLY USED BECAUSE AN
                        ENTITY_TYPE WHICH COMPOSES THE CLASS IS
                        USED. *).
```

Table 6.1 Examples of RADX Commands For Requirements Analysis (Continued)

COMMANDS TO TEST INFORMATION LOCALITY SPECIFICATIONS

```
HIERARCHY FILE_TRACE = FILE CONTAINS DATA;
                          DATA INCLUDES DATA.
HIERARCHY MESSAGE_TRACE = MESSAGE MADE BY FILE;
                       MESSAGE MADE BY DATA;
                       DATA INCLUDES DATA.
HIERARCHY ENTITY_TRACE = ENTITY_CLASS ASSOCIATES FILE;
                       FILE CONTAINS DATA;
                       DATA INCLUDES DATA;
                       ENTITY_CLASS ASSOCIATES DATA;
                       ENTITY_CLASS COMPOSED OF ENTITY_TYPE;
                       ENTITY_TYPE ASSOCIATES FILE;
                       ENTITY_TYPE ASSOCIATES DATA.
SET ENTITY_MEMBERS = ALL IN HIERARCHY ENTITY_TRACE.
SET LOCAL_ENTITY_MEMBERS = ENTITY_MEMBERS WITH LOCALITY LOCAL
                       (* THE LOCALITY OF ALL ENTITY RELATED INFORMATION
                          MUST BE GLOBAL. *).
SET MESSAGE_MEMBERS = ALL IN HIERARCHY MESSAGE_TRACE.
SET GLOBAL_MESSAGE_MEMBERS = MESSAGE_MEMBERS WITH LOCALITY GLOBAL
                       (* THE LOCALITY OF ALL MESSAGE RELATED INFORMATION
                          MUST BE LOCAL. *).
SET FILES_NOT_IN_MESSAGE = FILE WITHOUT MAKES.
SET FILES_NOT_IN_ENTITY = FILE THAT IS NOT ASSOCIATED.
SET INDEPENDENT_FILE = FILES_NOT_IN_MESSAGE AND FILES_NOT_IN_ENTITY.
SET GLB_DEFAULT_FILE = INDEPENDENT_FILE WITHOUT LOCALITY.
SET GLB_SPECIFIED_FILE = INDEPENDENT_FILE WITH LOCALITY = GLOBAL.
SET GLOBAL_FILE = GLB_DEFAULT_FILE OR GLB_SPECIFIED_FILE.
SET LOCAL_FILE = INDEPENDENT_FILE MINUS GLOBAL_FILE.
SET GLB_FILE_TRACE = DATA OR GLOBAL_FILE.
SET LOC_FILE_TRACE = DATA OR LOCAL_FILE.
SET GLOBAL_FILE_MEMBERS = GLB_FILE_TRACE IN HIER FILE_TRACE.
SET LOCAL_DATA_IN_GLOBAL_FILE = GLOBAL_FILE_MEMBERS WITH
                       LOCALITY = LOCAL
                       (* THE LOCALITY OF THE MEMBERS OF A GLOBAL
                          FILE MUST NOT BE LOCAL. *).
SET LOCAL_FILE_MEMBERS = LOC_FILE_TRACE IN HIER FILE_TRACE.
SET GLOBAL_DATA_IN_LOCAL_FILE = LOCAL_FILE_MEMBERS WITH
                       LOCALITY = GLOBAL
                       (* THE LOCALITY OF THE MEMBERS OF A LOCAL
                          FILE MUST NOT BE GLOBAL. *).
```

COMMANDS TO TEST DATA TYPE AND USE SPECIFICATIONS

```
SET ENUM = DATA WITH TYPE ENUMERATION.
SET MISSING_RANGE = ENUM WITH NO RANGE
                       (* A DATA WITH TYPE ENUMERATION MUST HAVE AN
                          INSTANCE OF THE RANGE ATTRIBUTE. *).
SET RNG = DATA WITH RANGE.
SET NO_TYPE = RNG WITH NO TYPE.
SET WRONG_TYPE = RNG WITH TYPE <> ENUMERATION.
SET DATA_NEEDING_TYPE_ENUMERATION = WRONG_TYPE OR NO_TYPE
                       (* ALL DATA WITH A RANGE ATTRIBUTE MUST HAVE
                          A TYPE AND THE TYPE MUST BE ENUMERATION. *).
```

Table 6.1 Examples of RADX Commands For Requirements Analysis (Continued)

COMMANDS TO TEST DATA TYPE AND USE SPECIFICATIONS (CONTINUED)

```
SET LOWEST_DATA = DATA WITHOUT INCLUDES DATA.
SET MISSING_TYPE = LOWEST_DATA WITHOUT TYPE
                   (* ALL REQUIREMENTS LEVEL DATA MUST HAVE A
                      SPECIFIED TYPE. *).
SET GAMMA_DATA = DATA WITH USE = GAMMA.
SET BOTH_DATA = DATA WITH USE = BOTH.
SET GAMMA_DATA = GAMMA_DATA OR BOTH_DATA.
SET INCORRECT_GAMMA_DATA = GAMMA_DATA THAT INCLUDES DATA
                   (* DATA WITH USE GAMMA CANNOT INCLUDE OTHER
                      DATA. *).
SET MISSING_USE_ATTRIBUTE = LOWEST_DATA WITHOUT USE
                   (* ALL LOWEST LEVEL DATA SHOULD HAVE USE = GAMMA
                      OR USE =BOTH. *).
SET INCORRECT_BETA_DATA = LOWEST_DATA WITH USE BETA
                   (* LOWEST LEVEL DATA CANNOT HAVE USE BETA. *).
```

COMMANDS TO TEST ORIGINATING_REQUIREMENTS AND VALIDATION SPECIFICATIONS

```
SET REQUIREMENTS = ORIGINATING_REQUIREMENT OR DECISION.
SET NOT_DECOMPOSED = REQUIREMENTS WITH NO TRACES
                   (* ALL ORIGINATING_REQUIREMENTS AND DECISIONS
                      MUST BE DECOMPOSED BY TRACING TO OTHER
                      ELEMENTS FOR THE REQUIREMENTS TO BE
                      COMPLETE. *).
SET NON_CONSTRAINING_PER_REQ = PERFORMANCE_REQUIREMENT WITHOUT
                               CONSTRAINS VALIDATION_PATH
                   (* A PERFORMANCE_REQUIREMENT MUST CONSTRAIN
                      A VALIDATION_PATH. *).
SET INCOMPLETE_PER_REQ = PERFORMANCE_REQUIREMENT WITHOUT TEST
                   (* FOR A PERFORMANCE REQUIREMENT TO TO BE
                      COMPLETE IT MUST HAVE A TEST. *).
SET NETS = R_NET OR SUBNET.
SET UNUSED_VAL_PT = VALIDATION_POINT WHICH IS NOT REFERRED TO
                   BY VALIDATION_PATH
                   (* A VALIDATION_POINT MUST BE USED BY AT
                      LEAST ONE VALIDATION_PATH. *).
SET MULTI_PROBING_VAL_POINTS = VALIDATION_POINT THAT IS MULTIPLE
                   REFERRED TO BY NETS
                   (* A VALIDATION_POINT CAN ONLY OCCUR ONCE
                      IN A NET STRUCTURE. *).
```

6.2 LISTING REQUIREMENTS

This section describes the various ways that RADX can be used to produce requirements documentation. The disposition of the output produced by RADX is determined by the following:

1) If a LIST command is input by the user, then the output from the statement is routed to the printer, ANAGRAPH, or both as specified by the control selected via REVS EXECUTIVE RCL (See Section 4).

2) If a PUNCH command is used, the printed output from the command is the same as that produced using the LIST command, and additionally, punched cards are generated that contain all printed images except blank lines and RADX error messages. The actual disposition of the punched cards is determined by a parameter of the REVSXQT macro as described in Section 9.

3) If a PLOT command is used, the command produces graphic displays on the CALCOMP plotter of the STRUCTUREs of those elements being plotted.

The examples in this section will rely mostly on the LIST command to illustrate the use of RADX to produce documentation, but the user is reminded that the keywords LIST and PUNCH can be interchanged to vary the disposition of the documentation.

6.2.1 Listing Sets

All LIST operations use one set as their operand. The simple syntax of the command is:

$$\left\{\begin{matrix}\text{LIST}\\ \text{PUNCH}\end{matrix}\right\}_{1}^{1} \text{<set identifier>.}$$

The set identifier may designate any set that is defined prior to the use of the LIST statement. As shown in the following examples, set identifier can designate the universal set ALL or ANY, an element type set, an element set, or a user defined set.

```
LIST ALL.
LIST DATA.
LIST UPDATE_STATE.
LIST SET X.
```

When a set is LISTed the members are displayed alphabetically by element type and within each element type by element name. After each element is displayed, associated attributes, relationships, and structures are displayed. Should the user want to vary or eliminate the associated information displayed about an element, the APPEND statement, described in Section 6.2.2, may be used.

The requirements information produced using this LIST command is in an indented format that contains legal RSL syntax. An example output is presented in Figure 6-1.

6.2.2 Selecting Associated Information to be Displayed

The APPEND command is used to specify the associated attributes, relationships, and structures that should be displayed following the display of an element. The syntax of the statement is:

$$\text{APPEND <type identifier>} \left\{\text{<append item>}\right\}_1^n .$$

In the statement, type identifier is an RSL element type name, the keyword ANY, or the keyword ALL and indicates the element type or element types to which the append item list applies. When ALL or ANY is specified, the list is applied to all element types in the ASSM. The following is a list of legal append items and the information that it causes to be appended to an element that is a subset of type identifier.

relation-name	- a particular RSL relationship.
attribute-name	- a particular RSL attribute.
REFERS	- elements that appear on the structure of the subject element.
REFERRED	- elements with structures that use the subject element.
ALL	- all attributes in alphabetical order, followed by all primary relationships in alphabetical order, followed by REFERs, followed alphabetically by all complementary relationships, followed by REFERRED, and finally the element STRUCTURE or PATH.
NONE	- no associated information.
STRUCTURE	- R_NET, SUBNET, or VALIDATION_PATH structure.

```
(RADX COMMAND=
LIST DATA.
------------

    DATA:  ACCEPTANCE_THRESHOLD,
         LOCALITY: LOCAL,
         TYPE: REAL,
         USE: GAMMA,
         INCLUDED IN:
               DATA:  T1_T2_GATE_DATA
               DATA:  T3_GATE_DATA.

    DATA:  ACCOUNTED_FOR,
         INITIAL_VALUE: NEITHER,
         LOCALITY: GLOBAL,
         RANGE: "NEITHER,COUNTED,SUMMED",
         TYPE: ENUMERATION,
         USE: BOTH,
         ASSOCIATED WITH:
               ENTITY_TYPE:  LOST_PULSE
               ENTITY_TYPE:  RETURNED_PULSE,
         OUTPUT FROM:
               ALPHA:  SET_COUNTED
               ALPHA:  SET_SUMMED,
         REFERRED BY:
               SUBNET:  SUM_RETURNS
               SUBNET:  TALLY_RETURNS,

    DATA:  ALPHA_ERROR,
         LOCALITY: LOCAL,
         TYPE: REAL,
         USE: GAMMA,
         INCLUDED IN:
               DATA:  T1_T2_RECEIVE
               DATA:  T3_RECEIVE,

    DATA:  ALPHA_PHASE_TAPER,
         LOCALITY: LOCAL,
         TYPE: REAL,
         USE: GAMMA,
         INCLUDED IN:
               DATA:  T1_T2_TRANSMIT
               DATA:  T3_TRANSMIT,
```

Figure 6-1 Sample Output From Standard LIST SET Command

ATTRIBUTE	- all attributes in alphabetical order.
RELATION RELATIONSHIP	- all primary relationships in alphabetical order followed by all complementary relationships in alphabetical order.
PRIMARY	- all primary relationships in alphabetical order.
COMPLEMENTARY	- all complementary relationships in alphabetical order.

When RADX is initially activated the append item for all elements is initialized to ALL. If a new APPEND statement is entered for a type identifier, the append item list replaces the previous list for the type identifier and remains active until a new APPEND statement changes the selection or RADX is terminated.

The order that append items appear in the APPEND statement determines the order that associated information is displayed about an element. A duplicate item in the statement will cause duplicate information to be displayed for the elements of the type indicated by the type identifier.

Examples of the use of the APPEND statement and the results produced using it with the LIST statement are provided in Figure 6-2.

6.2.3 Listing By Hierarchies

The LIST by HIERARCHY command provides a technique to vary the format and order of the display of set members. The syntax of the command is:

$$\left\{\begin{matrix}\text{LIST}\\ \text{PUNCH}\end{matrix}\right\}_1^1 \text{<set identifier> <positive connector>} \left\{\begin{matrix}\text{HIER}\\ \text{HIERARCHY}\end{matrix}\right\}_1^1$$

$$\text{hierarchy-name} \left[\text{<positive connector>} \left\{\begin{matrix}\text{MAP}\\ \text{SEQUENCE}\\ \text{GROUP}\end{matrix}\right\}\right]_1^1 .$$

In the statement, positive connector can be any of the terms described in Section 6.1.3. The hierarchy name identifies a hierarchy that must be defined using the define hierarchy capability described in Section 6.1.6. The set identifier specifies the subject set from which elements can be selected to be displayed. The optional part of the statement is called the type display option. If one of the options is not explicitly selected, then the MAP option is used to make the display. Each of the

```
[RADX COMMAND=
APPEND FILE:  CONTAINS.
--------------------------

[RADX COMMAND=
LIST FILE.
-----------

    FILE:  CANDIDATE.
         CONTAINS:
              DATA:  CANDIDATE_ENERGY
              DATA:  CANDIDATE_IMAGE_ID
              DATA:  CANDIDATE_WAVEFORM
              DATA:  PRIORITY.

    FILE:  COMMAND.
         CONTAINS:
              DATA:  COMMAND_ENERGY
              DATA:  COMMAND_IMAGE_ID
              DATA:  COMMAND_WAVEFORM
              DATA:  START_TIME.

    FILE:  STATE_FILE.
         CONTAINS:
              DATA:  STATE_DATA
              DATA:  STATE_ID.

    FILE:  TERMINATOR.
         CONTAINS:
              DATA:  DROP_REASON
              DATA:  DROP_TIME.

    FILE:  T1_T2_DATA.
         CONTAINS:
              DATA:  T1_T2_RECORD.

    FILE:  T1_T2_GATE.
         CONTAINS:
              DATA:  T1_T2_GATE_DATA.

    FILE:  T1_T2_WINDOW.
         CONTAINS:
              DATA:  T1_T2_WINDOW_DATA.

    FILE:  T3_DATA.
         CONTAINS:
              DATA:  T3_RECORD.

    FILE:  T3_GATE.
         CONTAINS:
              DATA:  T3_GATE_DATA.
```

Figure 6-2 Sample Use of APPEND and LIST Commands

```
(RADX COMMAND=
APPEND R_NET:  STRUCTURE.
-------------------------

(RADX COMMAND=
LIST R_NET.
-----------

   R_NET:  CC_RESPONSE.
        STRUCTURE:
           INPUT_INTERFACE:  CC_IN
           ALPHA:  VALIDATE_HEADER
           DO
               ALPHA:  ACKNOWLEDGE
               OUTPUT_INTERFACE:  CC_OUT
           AND
               CONSIDER  DATA:  COMMAND_ID
               IF (HANDOVER_IMAGE)
                   ALPHA:  TRACK_INITIATE
                   VALIDATION_POINT:  C2_IMAGE_HANDOVER
                   EVENT:  ALLOCATE
                   OUTPUT_INTERFACE:  DATA_RECORD
               OR (INITIATE_ENGAGEMENT_MODE)
                   ALPHA:  STARTER
                   VALIDATION_POINT:  STARTING_POINT
                   ALPHA:  ENGAGEMENT_INITIATION
                   EVENT:  SCHEDULE
                   EVENT:  SUMMARIZE
                   TERMINATE
               OR (TERMINATE_ENGAGEMENT_MODE)
                   ALPHA:  TERM_ENGAGEMENT
                   TERMINATE
               OR (DROP_TRACK)
                   SELECT ENTITY_CLASS:  IMAGE SUCH THAT (IMAGE_ID=HO_ID)
                   IF (FOUND)
                       SUBNET:  RECORD_DROP
                       ALPHA:  TERM_TRACK
                       OUTPUT_INTERFACE:  DATA_RECORD
                   OTHERWISE
                       ALPHA:  CC_ERROR_PROCESSING
                       TERMINATE
                   END
               OR (CC_MESSAGE_ERROR)
                   ALPHA:  CC_ERROR_PROCESSING
                   TERMINATE
               END
           END
        END.
```

Figure 6-2 Sample Use of APPEND and LIST Commands (Continued)

display options are explained below using the following hierarchy definition:

```
HIER INFO_SOURCE = INPUT_INTERFACE PASSES MESSAGE;
                   MESSAGE MADE BY FILE;
                   MESSAGE MADE BY DATA;
                   FILE CONTAINS DATA;
                   DATA INCLUDES DATA.
```

MAP Display Option

Either of the following statements can be used to select this option:

```
LIST ALL BY HIER INFO_SOURCE.

LIST ALL BY HIER INFO_SOURCE BY MAP.
```

An example of the display generated by this option is given in Figure 6-3. The following rules apply when listing by this option.

- An element is displayed if and only if it is a member of the set identified by set identifier and it is encountered while traversing the hierarchy.
- Elements are displayed in the order that they are traversed in the hierarchy.
- The syntax of the display is a special form that is not legal RSL syntax.
- Elements are indented according to the level in the hierarchy where they are encountered.
- The only associated information displayed about an element is the binding relationships between the elements in the hierarchy. Thus, the APPEND selection has no affect when listing with this option.

SEQUENCE Display Option

An example of the specification of this statement and the resulting display is provided in Figure 6-4. The following rules are applied to determine the format and content of displays generated when this option is selected.

- An element is displayed if and only if it is a member of the set identified by set identifier and it is encountered while the hierarchy is traversed.

```
(RADX COMMAND=
LIST ALL BY HIER INFO_SOURCE BY MAP.
------------------------------------
   INPUT_INTERFACE:  CC_IN
      PASSES
         MESSAGE:  HANDOVER
            MADE BY
               DATA:  COMMAND_ID
               DATA:  HO_ID
               DATA:  INITIAL_COVARIANCE
               DATA:  INITIAL_STATE
         MESSAGE:  MODE_CHANGE
            MADE BY
               DATA:  COMMAND_ID
         MESSAGE:  TERMINATION
            MADE BY
               DATA:  COMMAND_ID
               DATA:  HO_ID
   INPUT_INTERFACE:  RADAR_CLOCK_IN
      PASSES
         MESSAGE:  R_CLOCK_MESSAGE
            MADE BY
               DATA:  RADAR_CLOCK_TIME
   INPUT_INTERFACE:  RADAR_IN
      PASSES
         MESSAGE:  T1_T2_RETURN
            MADE BY
               FILE:  T1_T2_DATA
                  CONTAINS
                     DATA:  T1_T2_RECORD
                        INCLUDES
                           DATA:  NOISE_LEVEL
                           DATA:  RANGE_MARK_INFORMATION
                              INCLUDES
                                 DATA:  RANGE_MARK_TIME
                                 DATA:  SIGNAL_AMPLITUDE
            MADE BY
               DATA:  RADAR_TYPE
               DATA:  RR_ORDER_ID
               DATA:  T1_T2_RECEIVE
                  INCLUDES
                     DATA:  ALPHA_ERROR
                     DATA:  BETA_ERROR
                     DATA:  T1T2RTN_ERROR_REPORT
                        INCLUDES
                           DATA:  REASON_FOR_TRANSMISSION_FAILURE
                     DATA:  WAKE_ARRAY
                        INCLUDES
                           DATA:  AVERAGE_SIGNAL_POWER
                           DATA:  THRESHOLD_DOWN_CROSSING_TIME
                           DATA:  THRESHOLD_UP_CROSSING_TIME
```

Figure 6-3 Sample LIST By MAP

```
(RADX COMMAND=
LIST ALL BY HIER INFO_SOURCE BY SEQUENCE.
------------------------------------------

   INPUT_INTERFACE:  CC_IN.
        CONNECTS TO:
             SUBSYSTEM:  SSC2.
        ENABLES:
             R_NET:  CC_RESPONSE.
        PASSES:
             MESSAGE:  HANDOVER
             MESSAGE:  MODE_CHANGE
             MESSAGE:  TERMINATION.
        TRACED FROM:
             ORIGINATING_REQUIREMENT:
             TLS_DPSPR_PARAGRAPH_3_2_1_A_FUNCTIONAL_REQUIREMENTS
             ORIGINATING_REQUIREMENT:
             TLS_DPSPR_SUBSECTION_3_2_1_FUNCTIONAL_REQUIREMENTS.
        REFERRED BY:
             R_NET:  CC_RESPONSE.

   MESSAGE:  HANDOVER.
        MADE BY:
             DATA:  COMMAND_ID
             DATA:  HO_ID
             DATA:  INITIAL_COVARIANCE
             DATA:  INITIAL_STATE.
        PASSED THROUGH:
             INPUT_INTERFACE:  CC_IN.
        TRACED FROM:
             ORIGINATING_REQUIREMENT:
             TLS_DPSPR_PARAGRAPH_3_2_1_A_FUNCTIONAL_REQUIREMENTS
             ORIGINATING_REQUIREMENT:
             TLS_DPSPR_PARAGRAPH_3_2_1_B_FUNCTIONAL_REQUIREMENTS.

   DATA:  COMMAND_ID.
        LOCALITY: LOCAL.
        RANGE:
        "HANDOVER_IMAGE,DROP_TRACK,INITIATE_ENGAGEMENT_MODE,
         TERMINATE_ENGAGEMENT_MODE,CC_MESSAGE_ERROR".
        TYPE: ENUMERATION.
        USE: BOTH.
        MAKES:
             MESSAGE:  ACKNOWLEDGEMENT
             MESSAGE:  HANDOVER
             MESSAGE:  MODE_CHANGE
             MESSAGE:  TERMINATION.
        INPUT TO:
             ALPHA:  VALIDATE_HEADER.
```

Figure 6-4 Sample LIST By SEQUENCE

- Elements are displayed in the order that they are traversed in the hierarchy.
- The syntax of the display is the same as that produced by the standard LIST command which is an indented form of legal RSL syntax.
- Following the display of an element, associated information pertaining to the element is displayed as selected by the APPEND statement.

GROUP Display Option

Figure 6-5 gives an example of a statement selecting this option and the output generated in response to the statement. The rules that are used to determine the type of display produced when this option is selected are:

- An element is displayed if and only if it is a member of the set identified by set identifier and it is encountered while traversing the hierarchy.
- The syntax of the display is the same as that produced by the standard LIST command which is an indented form of legal RSL syntax.
- The APPEND option is applied following the display of an element to determine the associated information that should be displayed about the element.
- A group of elements encountered while traversing the hierarchy starting from an element in the top-of-the-hierarchy is displayed in an alphabetical order following the display of the top-of-the-hierarchy element.

A summary of the type of display produced by each of the options described in this section is given in Table 6.2.

6.2.4 Plotting Structures

This command is used to generate CALCOMP plots for those elements which have STRUCTUREs. The syntax of the statement is:

$$\text{PLOT <set identifier>} \left\{\text{<size selection>}\right\}_0^n .$$

The set identifier is used to identify the set of elements to be plotted, and of course, only those elements (if any) which have structures will be plotted. RADX uses the same basic technique for generating plots as that used by the RNETGEN function described in Section 5.2. Thus, if a structure has ANAGRAPH coordinates, the relative position of the structure symbols in the

```
(RADX COMMAND=
APPEND ALL PASSES, PASSED, MAKES, MADE BY, CONTAINS,
          CONTAINED, INCLUDES, INCLUDED.
-------------------------------------------------------

(RADX COMMAND=
LIST ALL BY HIER INFO_SOURCE BY GROUP.
-----------------------------------------

   INPUT_INTERFACE:  CC_IN.
        PASSES:
             MESSAGE:  HANDOVER
             MESSAGE:  MODE_CHANGE
             MESSAGE:  TERMINATION.

   DATA:  COMMAND_ID.
        MAKES:
             MESSAGE:  ACKNOWLEDGEMENT
             MESSAGE:  HANDOVER
             MESSAGE:  MODE_CHANGE
             MESSAGE:  TERMINATION.

   DATA:  HO_ID.
        MAKES:
             MESSAGE:  HANDOVER
             MESSAGE:  STATE_UPDATE
             MESSAGE:  TERMINATION
             MESSAGE:  TRACK_INITIATION
             MESSAGE:  TRACK_TERMINATION.

   DATA:  INITIAL_COVARIANCE.
        MAKES:
             MESSAGE:  HANDOVER.

   DATA:  INITIAL_STATE.
        MAKES:
             MESSAGE:  HANDOVER
             MESSAGE:  TRACK_INITIATION.

   MESSAGE:  HANDOVER.
        PASSED THROUGH:
             INPUT_INTERFACE:  CC_IN.
        MADE BY:
             DATA:  COMMAND_ID
             DATA:  HO_ID
             DATA:  INITIAL_COVARIANCE
             DATA:  INITIAL_STATE.

   MESSAGE:  MODE_CHANGE.
        PASSED THROUGH:
             INPUT_INTERFACE:  CC_IN.
        MADE BY:
             DATA:  COMMAND_ID.
```

Figure 6-5 Sample LIST By GROUP

Table 6.2 HIERARCHY Display Summary

DISPLAY / OPTION	SYNTAX	ORDER	INDENTATION	APPEND SELECTION APPLICABLE?
MAP	SPECIAL	AS ENCOUNTERED IN HIERARCHY	BY LEVEL IN HIERARCHY	NO
SEQUENCE	INDENTED RSL	AS ENCOUNTERED IN HIERARCHY	SAME AS STANDARD LIST COMMAND	YES
GROUP	INDENTED RSL	TOP-OF-HIERARCHY ELEMENT FOLLOWED ALPHABETICALLY BY ENCOUNTERED ELEMENTS	SAME AS STANDARD LIST COMMAND	YES

plot will be the same as their relative position when displayed on the ANAGRAPH. Should a structure not have coordinates, the automated plotting procedure is used for assigning (in the ASSM) the location of the structure symbols. The size of a plot is determined by the size selection specified by the user. The syntax of this part of the statement is given below followed by the rules that apply to the processing of the statement.

$$\left\{\begin{matrix}\text{WIDTH}\\ \text{HEIGHT}\end{matrix}\right\}_1^1 \text{ [=] value}$$

1. The value of the WIDTH parameter specifies in inches the width of the plot.

2. The value of the HEIGHT parameter specifies in inches the height of the plot.

3. The value of the WIDTH or HEIGHT parameter can be a real or integer number that is greater than zero. If the value of either parameter is less than or equal to zero, an error message will be displayed and no further action will be taken by RADX.

4. If the WIDTH parameter is not specified, then 8.0 is used for the parameter value.

5. If the HEIGHT parameter is not specified, then 10.0 is used for the value of the parameter.

6. If the specified value of the WIDTH parameter is greater than 50.0, a diagnostic message will be issued and 50.0 will be used in place of the specified value.

7. If the specified value of the HEIGHT parameter is greater than 29.0, a diagnostic message will be issued and 29.0 will be used in place of the specified value.

6.2.5 Listing RSL Descriptions

A description of the currently defined Requirements Statement Language (RSL) can be acquired from the ASSM by the LIST RSL statement which offers two basic formats for generating the description.

RSL Definition

The format of this display contains a syntax that is legal input to the RSL extension function (see Section 8). The syntax of the RADX statement to obtain the display is:

```
{LIST }1
{PUNCH}1 RSL [<display item>] .
```

A list of allowed forms for display item is given next with a description of the display generated when the display item is used.

element-type-name	- definition of the element type
relationship-name	- definition of the relationship
attribute-name	- definition of the attribute
ELEMENT_TYPE	- definition of all element types
RELATION RELATIONSHIP	- definition of all relationships
ATTRIBUTE	- definition of all attributes
ALL	- definition of all element types, relationships, and attributes. (Assumed value when display item is not specified.)

Examples of the use of this statement and the displays generated by it are contained in Figure 6-6. The RSL definitions in Appendix D.3 of this document were generated using the command:

LIST RSL.

<u>RSL Summary</u>

This display provides a summary of the legal uses of an element type for writing RSL. The summary is not a form that is acceptable to the RSL translator. The general syntax of the statement that invokes the summary is given below and examples are presented in Figure 6-7.

```
{LIST }1
{PUNCH}1 RSL [element-type-name] SUMMARY.
```

In the statement, the element type name identifies a particular RSL element type that should be summarized. If an element type is not specified in the statement, then a summary for all element types is generated. Appendix D.4 of this document was generated using the command:

LIST RSL SUMMARY.

```
[RADX COMMAND=
LIST RSL.
----------

   ELEMENT_TYPE:  ALPHA
                  (* A PROCESSING STEP IN THE FUNCTIONAL REQUIREMENTS
                     DOMAIN. *).
      STRUCTURE APPLICABILITY:  NET.

   ELEMENT_TYPE:  DATA
                  (* A SINGLE ITEM OR SET OF DATA THAT IS SPECIFIED
                     AND THAT WILL EITHER BE REQUIRED IN THE
                     REAL-TIME SOFTWARE OR IS NEEDED FOR
                     DESCRIPTIVE PURPOSES. *).

   ELEMENT_TYPE:  DECISION
                  (* THE DECISION THAT HAS BEEN MADE TO ENABLE
                     REQUIREMENTS TO BE TAKEN FROM THE DPSPR TO THE PPR.
                     THIS MEANS THAT THE REQUIREMENTS ARE NOT SIMPLY
                     ALLOCATED, BUT HAVE BEEN SUBJECTED TO
                     DERIVATION. *).

[RADX COMMAND=
LIST RSL ENTERED_BY.
--------------------

    ATTRIBUTE:  ENTERED_BY.
       APPLICABLE ELEMENT_TYPE:  ALPHA
                                 DATA
                                 DECISION
                                 ENTITY_CLASS
                                 ENTITY_TYPE
                                 EVENT
                                 FILE
                                 INPUT_INTERFACE
                                 MESSAGE
                                 ORIGINATING_REQUIREMENT
                                 OUTPUT_INTERFACE
                                 PERFORMANCE_REQUIREMENT
                                 R_NET
                                 SOURCE
                                 SUBNET
                                 SUBSYSTEM
                                 UNSTRUCTURED_REQUIREMENT
                                 VALIDATION_PATH
                                 VALIDATION_POINT
                                 VERSION.
       VALUE: TEXT
                   (* THE IDENTITY OF THE LAST PERSON TO ENTER
                      INFORMATION ABOUT THE ELEMENT. *).
```

Figure 6-6 Sample LIST RSL Definition

```
[RADX COMMAND=
LIST RSL SUMMARY.
-----------------

    ELEMENT_TYPE:  ALPHA
      LEGAL RELATIONSHIPS:
        CREATES:
            ENTITY_CLASS
        DESTROYS:
            ENTITY_CLASS
        FORMS:
            MESSAGE
        IMPLEMENTS:
            VERSION
        INPUTS:
            DATA
            FILE
        OUTPUTS:
            DATA
            FILE
        SETS:
            ENTITY_TYPE
        DOCUMENTED ("BY"):
            SOURCE
        EQUATED ("TO"):
            SYNONYM
        TRACED ("FROM"):
            DECISION
            ORIGINATING_REQUIREMENT
      LEGAL ATTRIBUTES:
        ARTIFICIALITY:
            ARTIFICIAL
            VALIDATION
            IMPLEMENT_APPROXIMATELY
            IMPLEMENT_PRECISELY
        BETA:
            TEXT
        COMPLETENESS:
            INCOMPLETE
            COMPLETE
            CHANGEABLE
        DESCRIPTION:
            TEXT
        ENTERED_BY:
            TEXT
        GAMMA:
            TEXT

    ELEMENT_TYPE:  DATA
      LEGAL RELATIONSHIPS:
        •
        •
        •
```

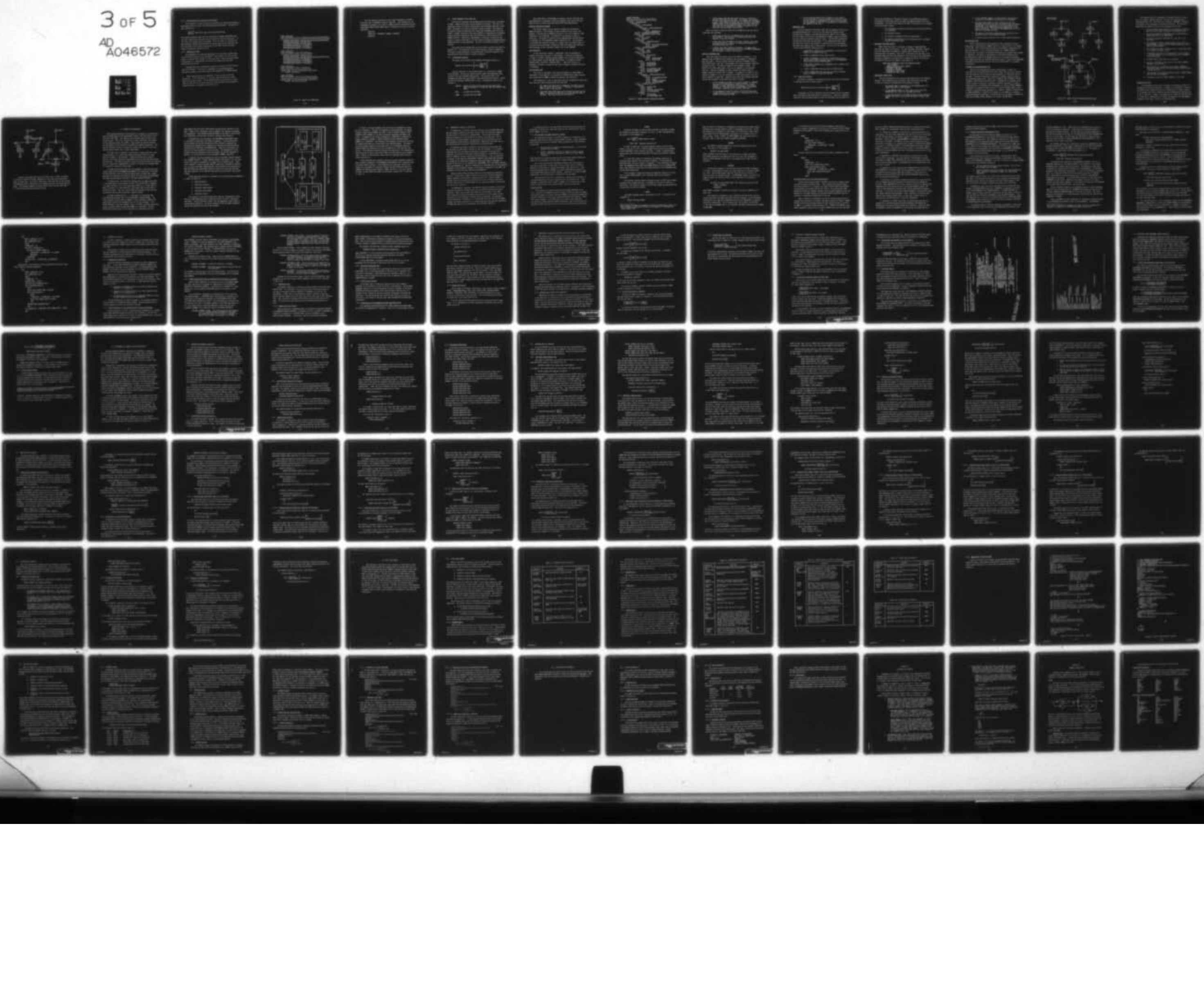
Figure 6-7 Sample LIST RSL SUMMARY

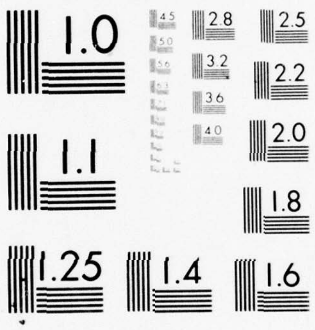
```
[RADX COMMAND=
LIST RSL DATA SUMMARY.
----------------------

   ELEMENT_TYPE:  DATA
      LEGAL RELATIONSHIPS:
        DELAYS:
             EVENT
        IMPLEMENTS:
             VERSION
        INCLUDES:
             DATA
        MAKES:
             MESSAGE
        ORDERS:
             FILE
        ASSOCIATED ("WITH"):
             ENTITY_CLASS
             ENTITY_TYPE
        CONTAINED ("IN"):
             FILE
        DOCUMENTED ("BY"):
             SOURCE
        EQUATED ("TO"):
             SYNONYM
        INCLUDED ("IN"):
             DATA
        INPUT ("TO"):
             ALPHA
        OUTPUT ("FROM"):
             ALPHA
        RECORDED ("BY"):
             VALIDATION_POINT
        TRACED ("FROM"):
             DECISION
             ORIGINATING_REQUIREMENT
      LEGAL ATTRIBUTES:
        ARTIFICIALITY:
             ARTIFICIAL
             VALIDATION
             IMPLEMENT_APPROXIMATELY
             IMPLEMENT_PRECISELY
        COMPLETENESS:
             INCOMPLETE
             COMPLETE
             CHANGEABLE
        DESCRIPTION:
             TEXT
        ENTERED_BY:
             TEXT
        INITIAL_VALUE:
             NAMED
             NUMERIC
```

Figure 6-7 Sample LIST RSL SUMMARY (Continued)

AD-A046 572 TRW DEFENSE AND SPACE SYSTEMS GROUP HUNTSVILLE ALA F/G 9/2
REVS USERS MANUAL. SREP FINAL REPORT. VOLUME II.(U)
AUG 77 M E DYER, L J GUNTHER, R W SMITH DASG60-75-C-0022
UNCLASSIFIED TRW-27332-6921-026-VOL-2 NL

3 OF 5
AD A046572




1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

6.2.6 Listing Control and Extension Permissions

This statement is used to obtain a listing of the control and extension permissions that are active for the ASSM being used by REVS. The syntax of the statement is:

$$\left\{\begin{matrix}\text{LIST}\\\text{PUNCH}\end{matrix}\right\}_1^1 \text{ PERMISSION GIVEN control-permission-name.}$$

The control permission name must be the name of a control permission that has been established using the extension portion of the RSL translator. If it is not, the command will be rejected. If it is, RADX will display the RSL statement CONTROL_PERMISSION with the given name and will generate the RSL statements for all other permissions in the ASSM, should any exist. Figure 6-8 contains examples of this.

6.2.7 Punching the ASSM

As indicated in the previous portions of this section, anything that can be LISTed by RADX can also be PUNCHed. This allows a data base to be stored in "source form" instead of "data base form", two or more data bases to be merged, or a requirements data base to be moved from one REVS installation to another.

The REVS Executive and RADX have special provisions to allow information to be PUNCHed. The value of a parameter of the REVSXQT macro as described in Section 9 actually controls the disposition of PUNCHed images.

After information has been PUNCHed, it can be input to the RSL Translator. If the information is on cards, then obviously the cards serve as the input source. If the information has been placed on a file, then the file can be the input source by using the REVS Executive ADDFILE capability that is documented in Section 4.2.2.

```
[RADX COMMAND=
LIST PERMISSION GIVEN CHANGE_AND_EXTENSION_CONTROLLER,
-------------------------------------------------------------
    CONTROL_PERMISSION CHANGE_AND_EXTENSION_CONTROLLER,
    CONTROL_PERMISSION CONTROLLER_1,
    CONTROL_PERMISSION CONTROLLER_2,
    EXTENSION_PERMISSION EXTENDER_1,
    EXTENSION_PERMISSION EXTENDER_2,
    EXTENSION_PERMISSION EXTENDER_3,

[RADX COMMAND=
LIST PERMISSION GIVEN CONTROLLER_1,
-------------------------------------
    CONTROL_PERMISSION CONTROLLER_1,
    CONTROL_PERMISSION CHANGE_AND_EXTENSION_CONTROLLER,
    CONTROL_PERMISSION CONTROLLER_2,
    EXTENSION_PERMISSION EXTENDER_1,
    EXTENSION_PERMISSION EXTENDER_2,
    EXTENSION_PERMISSION EXTENDER_3,

[RADX COMMAND=
LIST PERMISSION GIVEN EXTENDER_1,
-----------------------------------
*ERROR 2580 ILLEGAL PERMISSION SPECIFIED,
-     SYMBOL EXTENDER_1

[RADX COMMAND=
LIST PERMISSION GIVEN NOT_A_NEED_TO_KNOW,
-------------------------------------------
*ERROR 2580 ILLEGAL PERMISSION SPECIFIED,
-     SYMBOL NOT_A_NEED_TO_KNOW
```

Figure 6-8 Sample LIST PERMISSION

The user should be cautious to only input information to the RSL translator that is PUNCHed by RADX as legal RSL. The following are the recommended statements to be used for PUNCHing the RSL definition and the requirements specification when they are intended to be input to the RSL translator.

```
PUNCH RSL.
APPEND ALL:  ATTRIBUTE, PRIMARY, STRUCTURE.
PUNCH ALL.
```

6.3 USING AUTOMATED STATIC ANALYSIS

This section describes the automated static analysis that is provided by RADX. There are two basic types available to the user. The first is a consistency check of critical relationships and attributes specified in the ASSM. The second is a data flow analysis within an R_NET. The selection to perform a data flow analysis also causes the consistency test to be performed. Another function of this area of RADX is to perform the consistency check and collect information from the ASSM during initialization of the SIMGEN function. There are no requirements on the user to cause this but there is a need to know about it so that certain information displayed during SIMGEN execution can be understood. This will be further explained in this section.

The actual error messages that can occur from an analysis are given in Appendix F.2. The discussion that follows provides a general statement of the types of errors detected, an interpretation of analysis displays, and a description of the RCL used to invoke an analysis.

6.3.1 Consistency Analysis

The syntax for activating the automated consistency analysis is:

$$\text{ANALYZE} <\text{set identifier}> \left[\text{USING} \begin{Bmatrix}\text{IMPLIED}\\ \text{BETA}\\ \text{GAMMA}\end{Bmatrix}\right].$$

The set identifier is the identification of a collection of R_NETs to be analyzed. The set may contain elements other than R_NETs but their inclusion will not influence the analysis. The optional part of the statement specifies the type of DATA to be used during the analysis. Should no option be selected, IMPLIED DATA is used. The meaning of each of the options is:

IMPLIED	- ignore the USE attribute and use the lowest level DATA in the ASSM (i.e., DATA that does not INCLUDE other DATA).
BETA	- use DATA with USE BETA.
GAMMA	- use DATA with USE GAMMA.

When USING BETA or USING GAMMA is selected, the USE attribute test described below is performed to identify anomalies that result from the specification of the USE attribute and conflicting relationships.

Analysis Information Network

The initial output from an analysis, either user activated or SIMGEN activated, is the display of the information to be used in the analysis. For the case of SIMGEN activation, it is also a display of the information to be simulated. An example of the display is given in Figure 6-9. The format is similar to that generated when the MAP option is selected in a LIST HIERARCHY command (Section 5.2.3). The exception is that when the character string, (*), appears after an element name, it indicates that the pertinent information about the element has previously been displayed in the information network and is not repeated.

Loop Detection

As the information network is generated a test is performed to identify loops that may occur in the network. A loop can be caused by a direct or indirect reference between SUBNETs and a recursive definition of a DATA element via the INCLUDES relationship. An example is DATA X INCLUDES DATA Y and DATA Y INCLUDES DATA X. When such an error is detected, a message is issued and the path of elements containing the loop is displayed.

USE Attribute Test

This test is performed if the option USING BETA or USING GAMMA is selected or if the analysis is activated from SIMGEN which always requires that either BETA or GAMMA be chosen for analysis and simulation.

The following is a list of the errors that can be detected by this test when they occur while analyzing DATA with USE BETA.

1. The lowest level DATA that is needed for the BETA analysis does not have a USE attribute. A message is issued and the DATA is used in the analysis.
2. The lowest level DATA required for the BETA analysis has USE GAMMA. A message indicates that it should have USE BETA or USE BOTH. The item is used for the analysis.

```
[RADX COMMAND=
ANALYZE DATA_FLOW ALL USING BETA.
*************************************
    R_NET: CC_RESPONSE
      REFERS TO
          ALPHA: ACKNOWLEDGE
            FORMS
              MESSAGE: ACKNOWLEDGEMENT
                MADE BY
                  DATA: COMMAND_ID
          ALPHA: CC_ERROR_PROCESSING
          ALPHA: ENGAGEMENT_INITIATION
            OUTPUTS
              DATA: MODE
          ALPHA: STARTER
            OUTPUTS
              FILE: WAVEFORM_TABLE
                CONTAINS
                  DATA: WF_CHARACTERISTICS
                  DATA: WF_NAME
          ALPHA: TERM_ENGAGEMENT
            OUTPUTS
              DATA: MODE
          ALPHA: TERM_TRACK
            OUTPUTS
              FILE: TERMINATOR
                CONTAINS
                  DATA: DROP_REASON
                  DATA: DROP_TIME
            INPUTS
              DATA: CLOCK_TIME
              DATA: DROP_REASON
              DATA: DROP_TIME
            OUTPUTS
              DATA: DATA_RECORD_TYPE
              DATA: REASON_FOR_DROP
              DATA: TIME_OF_DROP
            FORMS
              MESSAGE: TRACK_TERMINATION
                MADE BY
                  DATA: DATA_RECORD_TYPE
                  DATA: HO_ID
                  DATA: REASON_FOR_DROP
                  DATA: TIME_OF_DROP
          ALPHA: TRACK_INITIATE
            INPUTS
              DATA: CLOCK_TIME
              DATA: HO_ID
              DATA: INITIAL_COVARIANCE
              DATA: INITIAL_STATE
            OUTPUTS
              DATA: COVARIANCE
              DATA: DATA_RECORD_TYPE
```

Figure 6-9 Sample Analysis Information Network

3. A DATA element with USE BETA either directly or indirectly INCLUDES other DATA with USE BETA. The highest level element is used in the analysis and the user is informed of the problem. However, should there be another need for the lower level DATA item that does not go through the higher level item, the lower level item will be selected for analysis in addition to the previously selected higher level element.

When an analysis is done for DATA with USE GAMMA the test and actions listed next are performed.

1. A DATA item required for the GAMMA analysis does not have a USE attribute. The situation is reported to the user and the DATA is used for analysis.
2. A DATA item, with USE GAMMA or USE BOTH, INCLUDES other DATA. The appropriate error message is issued and the analysis is performed without the item.
3. A lowest level DATA element has USE BETA. A message which indicates that it should have USE BOTH or USE GAMMA is generated and the element is used in the analysis.

<u>LOCALITY Attribute Test</u>

This test checks the correct use of the LOCALITY attribute. It is concerned with the DATA and FILE elements that are members of Repetitive Data Sets (RDS). The RSL elements that make an RDS are MESSAGE, ENTITY_CLASS, ENTITY_TYPE, and FILE. The elements that can be assigned a LOCALITY are DATA and FILE. The value of LOCALITY can be GLOBAL or LOCAL. The LOCALITY of an RDS (other than a FILE) and its members is not optional. They are: LOCALITY of MESSAGE is LOCAL; LOCALITY OF ENTITY_CLASS is GLOBAL; LOCALITY of ENTITY_TYPE is GLOBAL. The LOCALITY of a DATA or FILE element is GLOBAL if it is not contained in an RDS that dictates its value and no LOCALITY is specified. Based on these facts, the following test and actions are performed in the order that they are listed:

1. If the information content (i.e., FILE or DATA) of a MESSAGE has an explicit declaration of LOCALITY GLOBAL, an error message is issued identifying the conflict. For analysis purposes, all the information in the MESSAGE is considered to be LOCAL, regardless of the LOCALITY attribute.
2. If the members of an ENTITY_CLASS or ENTITY_TYPE have been assigned LOCALITY LOCAL, the error is identified and all the members are treated as GLOBAL.

3. If the specified LOCALITY of the DATA in a FILE differs from the specified or defaulted LOCALITY of the FILE, then an error message is generated which identifies the problem and the LOCALITY of the DATA is considered to be the same as the LOCALITY of the FILE.

Membership Test

This test identifies those DATA items that are members of more than one Repetitive Data Set. An error of this nature can possibly cause multiple error messages to be produced about the LOCALITY of an element. For this reason, the errors of the membership category should be examined to see if they are triggering LOCALITY errors. The following summarizes the specific tests that are performed. It is indeterminant what effect the errors will have on the analysis or simulation of the requirements data base that contains any of these errors. An error will be identified if any of the following conditions occur:

1. A DATA MAKES a MESSAGE and is CONTAINED in a FILE but the MESSAGE is not MADE by the FILE.
2. A DATA is CONTAINED in more than one FILE.
3. A DATA is CONTAINED in a FILE and is ASSOCIATED with an ENTITY_TYPE or ENTITY_CLASS but the FILE is not ASSOCIATED with the ENTITY_TYPE or ENTITY_CLASS.
4. A DATA MAKES a MESSAGE and is ASSOCIATED with an ENTITY_TYPE or ENTITY_CLASS.
5. A DATA is ASSOCIATED with more than one ENTITY_CLASS.
6. A DATA is ASSOCIATED with more than one ENTITY_TYPE that does not COMPOSE the same ENTITY_CLASS.

6.3.2 Data Flow Analysis

The option to perform a consistency analysis and a data flow analysis is selected using the following syntax:

$$\text{ANALYZE DATA\_FLOW <set identifier>} \left[\text{USING} \begin{Bmatrix} \text{IMPLIED} \\ \text{BETA} \\ \text{GAMMA} \end{Bmatrix}_1^1 \right] .$$

The meaning of the set identifier and the optional part of the statement is the same as the description in Section 6.3.1. When this command is specified, the consistency analysis tests are performed and an analysis of the use and assignment of information based on the predecessor/successor relationships of the nodes within an R_NET and dependent SUBNET structures is performed. The actual tests made by the data flow analysis can be grouped into the following classes.

1. The incomplete or ambiguous specification of branch conditions in a structure.
2. Net structure errors.
3. The incorrect assignment and use of information.
4. The ambiguous identification of information that is assigned or used in parallel paths.

Walk-Back From Error Source

When a data flow error is detected, a message is generated that describes the nature of the error. For most messages, additional information that identifies the element or elements which caused the error is displayed after the message. After this, a walk-back from the node in the structure where the error was detected to the first node of the R_NET being analyzed is produced to aid the user in locating the source of the error. An example of how this will appear in the output is:

```
*ERROR 2664 INFORMATION ALWAYS USED BEFORE ASSIGNED.
        DATA: WLFE04.
      * ERROR DETECTED AT ALPHA: ALPHA7H
      * PRECEDED BY AND-NODE
      * PRECEDED BY AND-NODE
      * PRECEDED BY ALPHA: ALPHA7
      * PRECEDED BY R_NET: RNET7
```

Conditional Branch Test

The errors detected by this test are those which occur at a CONSIDER OR node. The following are the conditions that must hold for the node. If any are violated, an appropriate error message is issued.

1. The element that is the subject of the CONSIDER must be of element type DATA or ENTITY_CLASS.
2. If the CONSIDER element is DATA, then it must have TYPE ENUMERATION and a value for the RANGE attribute.
3. A value that appears in the RANGE attribute cannot duplicate another value and it must not be the same as a name in the ASSM.

4. If the CONSIDER element is an ENTITY_CLASS, there must be instances of the COMPOSED relationship for the element.

5. The values that appear in branch expressions from the CONSIDER node must be legal for the element being considered. For example, if an ENTITY_CLASS is being considered, only ENTITY_TYPEs which COMPOSE the ENTITY_CLASS can be in the branch expressions. For DATA of TYPE ENUMERATION, the value must be in the RANGE attribute.

6. The values in the branch expressions must include all the possible values of the considered element.

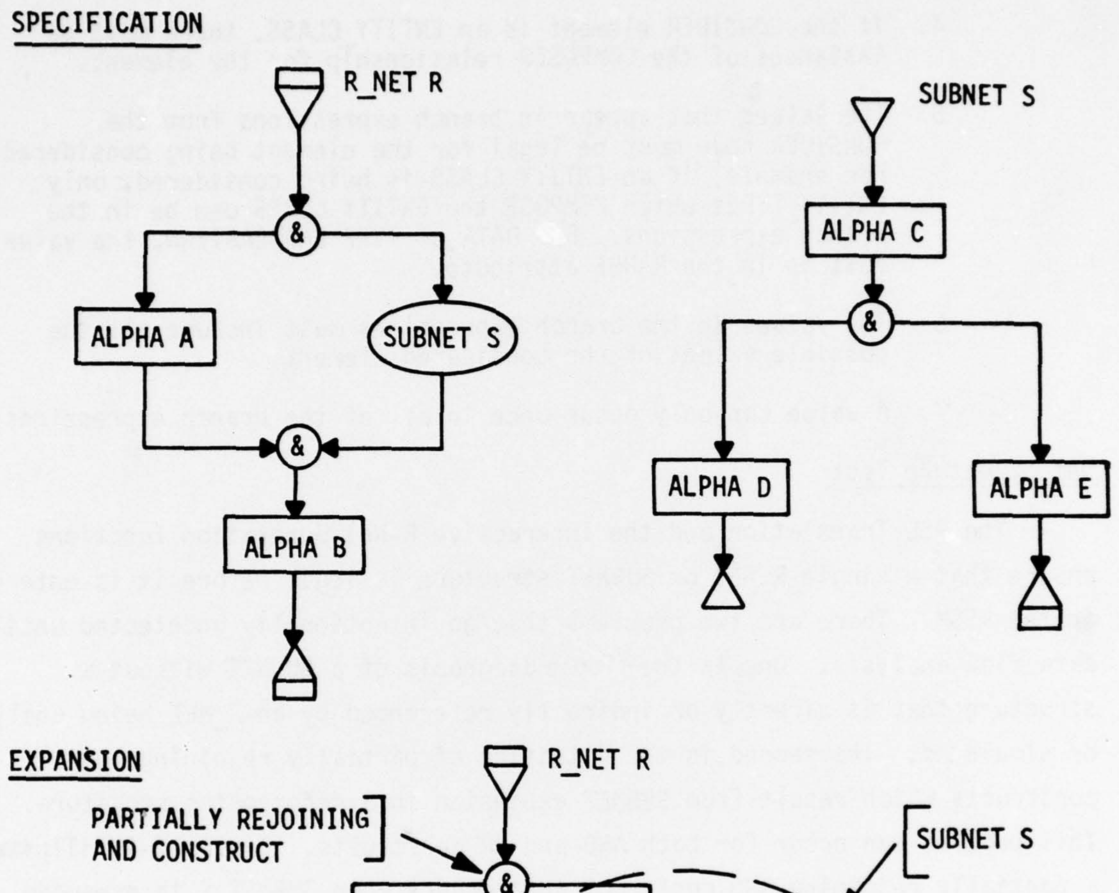
7. A value can only occur once in all of the branch expressions.

Net Structure Test

The RSL Translation and the Interactive R-Net Generation functions ensure that a single R_NET or SUBNET structure is legal before it is entered in the ASSM. There are two problems that go intentionally undetected until data flow analysis. One is the simple diagnosis of a SUBNET without a structure that is directly or indirectly referenced by an R_NET being analyzed or simulated. The second is the detection of partially rejoining logic constructs which result from SUBNET expansion in a referencing structure. This problem can occur for both AND and OR constructs. Figure 6-10 illustrates a partially rejoining AND construct that occurs when SUBNET S is expanded into R_NET R.

Information Assignment/Usage Test

Based on the predecessor/successor relationships of nodes in a structure, this test identifies errors that result from the use of information that does not have a source and the assignment of information that is not used. The term information is applied in a general sense to refer to the following element types: DATA, FILE, MESSAGE, INPUT_INTERFACE, OUTPUT_INTERFACE, ENTITY_CLASS, and ENTITY_TYPE. Information is considered to be used at a node if its value is required to compute information, to make a branch decision, to identify other information, or PASS an OUTPUT_INTERFACE. Information is assumed to have a source if it has an INITIAL_VALUE, exists external to the R-Net being analyzed (i.e., LOCALITY GLOBAL or acquired from an INPUT_INTERFACE), is OUTPUT by an ALPHA, or is a member of a properly maintained Repetitive Data Set. The operations used to maintain a Repetitive Data Set are FORM, CREATE, DESTROY, SET, SELECT, and FOR EACH.



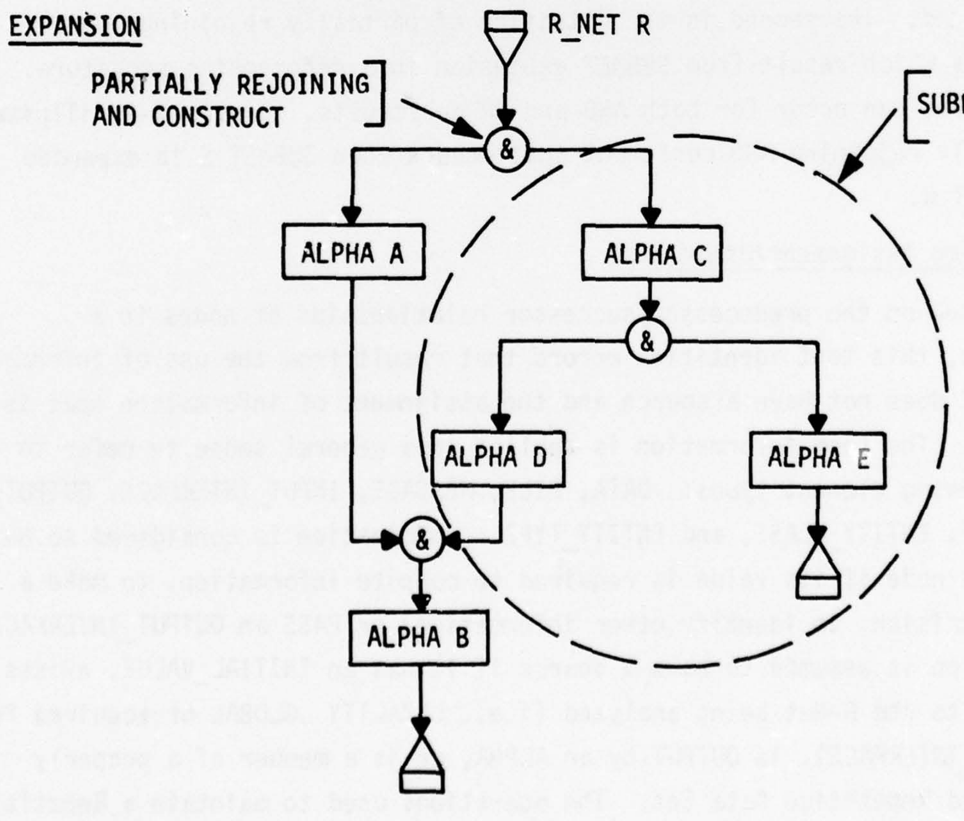


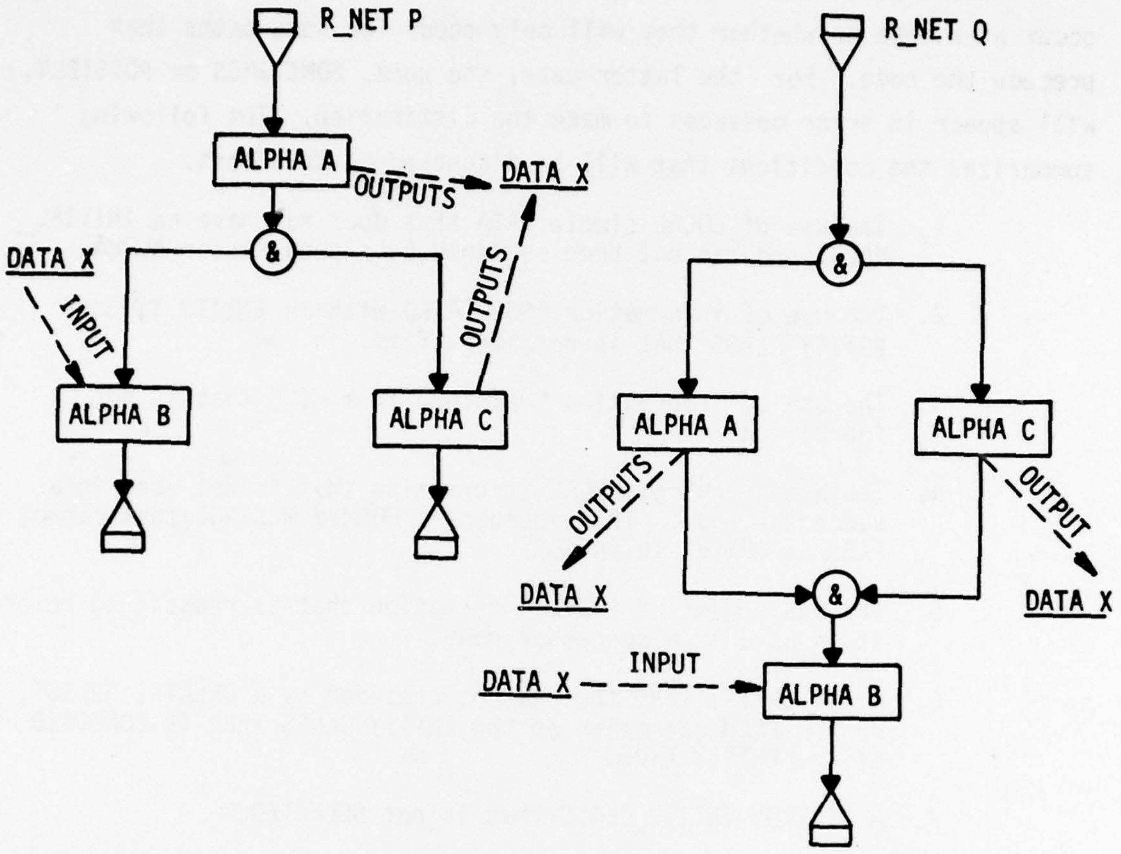
Figure 6-10 Sample Partially Rejoining AND-Construct

The detected errors are segmented as to whether they will always occur at a node or whether they will only occur for some paths that precede the node. For the latter case, the word, SOMETIMES or POSSIBLY, will appear in error messages to make the distinction. The following summarizes the conditions that will be diagnosed by this test.

1. The use of LOCAL simple DATA that does not have an INITIAL_VALUE and has not been assigned by a predecessor ALPHA.
2. The use of information ASSOCIATED with an ENTITY_TYPE or ENTITY_CLASS that is not identified.
3. The use of information CONTAINED in a FILE that is not identified.
4. The assignment of LOCAL information that is not used in a successor node. This includes a FORMed MESSAGE that cannot PASS an OUTPUT_INTERFACE.
5. The assignment of LOCAL information that is reassigned before it is used in a successor node.
6. A SET ENTITY_TYPE that is not preceded by a CREATE, SELECT, or FOR EACH operation on the ENTITY_CLASS that is COMPOSED of the ENTITY_TYPE.
7. A DESTROY ENTITY_CLASS that is not SELECTED.
8. The use of information from more than one INPUT_INTERFACE MESSAGE on the same path.
9. The selection of a new ENTITY_CLASS without a COMPOSED ENTITY_TYPE being SET for the current ENTITY_CLASS following a CREATE operation.
10. The traversal of an OUTPUT_INTERFACE without a FORMed MESSAGE that can PASS the INTERFACE.
11. Information passing an INPUT_INTERFACE is not used.

Ambiguous Flow Test

For the purpose of this test, an R_NET is considered to contain an ambiguous data flow specification if a change in the sequence of execution of parallel paths causes the source of information for a node to change. It is possible that such a condition has been intentionally specified and is not a true error. However, the ambiguity will always be identified and it is the user's responsibility to access the impact on the requirements. Examples of this type of error are illustrated below.



In both of the R_NETs the source for DATA X that is INPUT to ALPHA B can either be ALPHA A or ALPHA C depending on the order that the parallel paths are exercised. For R_NET P, RADX would inform the user of the problem by displaying the fact that DATA X is assigned and used on different parallel paths. The message triggered by R_NET Q would inform the user that DATA X is assigned from more than one parallel path.

7.0 SIMULATING REQUIREMENTS

REVS supports generation and execution of simulators of the data processing system based on the requirements stated in RSL and stored in the data base (the ASSM). The simulators are discrete event and are of two distinct types. The first, a functional or beta simulation, uses models of the processing steps, the ALPHAs. These models may employ shortcuts and use models of the true data to simulate the required processing. This type of simulation serves as a means to validate the overall flow of processing against higher level system requirements.

The other type of simulation, an analytic or gamma simulation, uses analytic models, i.e., models that employ algorithms similar to those which will appear in the software. A gamma simulation may be used to define a set of data processing algorithms which have the required accuracy and stability. This does not establish real-time feasibility of the algorithm set for a particular implementation; instead it provides an existence proof of an analytic solution to the problem.

For gamma simulations, REVS will also generate a simulation post processor containing the PERFORMANCE_REQUIREMENT TESTs specified in the ASSM. The post processor tests are executed against data recorded from a gamma simulation to establish the analytic feasibility that the set of algorithms will meet the performance required of the data processing system.

As stated above, the distinction between beta and gamma simulations is the ALPHA models and the degree to which the required DATA is modeled. In a gamma simulator, all of the requirements DATA (i.e., lowest level of DATA) is present; in a beta simulator, only those DATA with USE BETA or USE BOTH are used. Thus, for a given ASSM the remainder of the simulator (processing flow and data structures) remains constant.

The ASSM representation of the requirements is translated into simulator/post processor code in the programming language PASCAL. The flow structure of each R_NET and SUBNET is used to develop a PASCAL procedure whose control flow implements that of the net structure. Each ALPHA reference on the nets becomes a call to a PASCAL procedure containing either the model (BETA) or algorithm (GAMMA) for the ALPHA. The data definitions and structure for the simulator are synthesized from the requirements DATA and their relationships and attributes specified in the ASSM. Based on this information, data management and recording procedures are synthesized. This software generated based on the ASSM is consolidated with simulation utilities and a driver to construct the simulator.

In generating a post processor, the PERFORMANCE_REQUIREMENTs TESTs in the ASSM are translated into PASCAL procedures. The data definitions and structures for the post processor and the retrieval procedures for accessing data recorded by the simulator are synthesized from the ASSM relationships and attributes of RECORDED DATA. These are consolidated with control utilities to generate a post processor. The dependencies of the simulator and post processor on the RSL concepts are further described in Section 7.1, as are the rules for writing BETAs, GAMMAs, and TESTs.

A simulator and post processor are built by the REVS Simulator Generation (SIMGEN) function. Through SIMGEN commands (See Section 7.3), the user controls the type of simulation, its scope, and its identification. After first internally invoking the RADX ANALYZE capability, discussed in Section 6.3, to analyze the ASSM, SIMGEN performs the translation and consolidation described above and establishes conditions which will cause the simulator (and post processor) to be compiled (see Section 9.0 for the REVS job control stream).

A REVS generated simulator is composed of the following major components, as shown in Figure 7-1:

- R_NET procedures
- Simulation Executive
- Simulation Event Manager
- Simulation Data Manager
- Simulation driver.

The simulator is of the discrete event type and is designed to interface the R_NET procedures with a simulation driver to form a closed-loop simulation. Overall simulation control and the engagement clock reside in the Simulation Executive. The driver interfaces with the R_NET procedures through the Simulation Data and Event Managers.

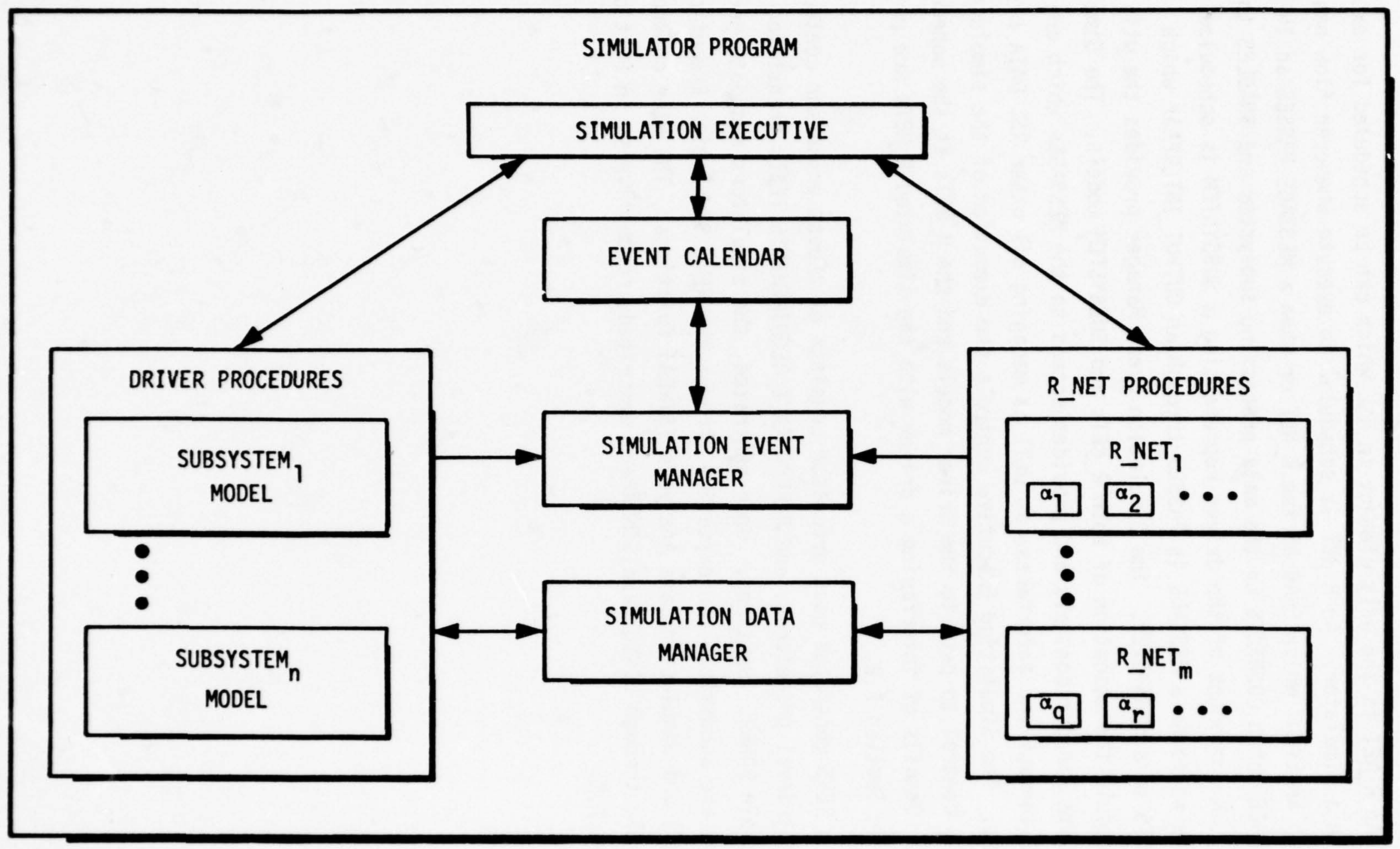


Figure 7-1 Simulator Functional Components

An R_NET is the only element in RSL which can be scheduled for execution in a simulator. An R_NET is scheduled to execute whenever flow passes through an EVENT which ENABLES the R_NET or when a MESSAGE PASSES an INPUT_INTERFACE which CONNECTS to the data processing subsystem and ENABLES the R_NET. A component of the driver representing a SUBSYSTEM is scheduled to execute whenever a MESSAGE is PASSED through an OUTPUT_INTERFACE which CONNECTS to a SUBSYSTEM. The Simulation Event Manager provides the utilities to schedule the execution of both R_NETs and SUBSYSTEM models. The Simulation Data Manager controls and provides access to the MESSAGEs which are PASSED through the interfaces, as well as managing all other RSL DATA constructs. The Simulation Executive controls the execution of the simulator by causing control to pass to the driver models and the R_NETs at the scheduled times. Details of interfacing a driver with the simulated R_NETs are provided in Section 7.2.

A REVS generated post processor consists of a post processor controller, data retrieval procedures, and PERFORMANCE_REQUIREMENTs TESTs translated into executable PASCAL functions. Once generated, the simulators and post processors are accessible for execution through the REVS Simulation Execution (SIMXQT) and Simulation Data Analysis (SIMDA) functions. The user controls available through SIMXQT and SIMDA are described, respectively, in Sections 7.4 and 7.5.

7.1 PREPARING AN ASSM FOR SIMULATION

In generating a simulator and post processor from the ASSM, REVS uses the RSL concepts comprising the Data, Alpha, R-Net and Validation Segments (see Sections 3.1 through 3.4, respectively). The concepts are either directly represented in the simulator or used in its generation. All of the concepts in these segments are pertinent with the exceptions of the DATA attributes MAXIMUM_VALUE, MINIMUM_VALUE, RESOLUTION, and UNITS and of the VALIDATION_PATH attributes MAXIMUM_TIME and MINIMUM_TIME.

To generate meaningful simulators, the conventions associated with the concepts (as explained in Section 3) must be followed. It is further necessary that the requirements pertinent to functional flow have been completed (for example, that each interface CONNECTS to a SUBSYSTEM, each R_NET is ENABLED, and each interface PASSES MESSAGEs). Also, as described in Section 5, the names for requirements elements should have been selected to conform to the installation dependent naming requirements described in Section 10 and so as not to conflict with BETA/GAMMA, TEST, and PASCAL reserved words (see Appendix B). REVS will always build a simulator even if the resulting simulator is not meaningful because of omissions or anomalies in the ASSM; the simulator will have the same omissions and anomalies.

Where possible, REVS provides defaults in the simulation generation process: if an R_NET or SUBNET structure is omitted, REVS will generate a "dummy" procedure representing an empty structure; if a BETA or GAMMA is omitted, REVS will provide one (see Section 7.2.1); if certain attributes of DATA such as TYPE or INITIAL_VALUE are not specified, default values will be used in the simulator.

In generating a simulator, the ANALYZE function of RADX is first automatically executed to analyze errors and to identify default action where possible. Where the user does not follow the conventions of RSL, RADX will attempt to diagnose the problem (see Section 6.3) but cannot correct the error; thus an erroneous simulator will be generated. Consequently, before generating a simulator, it is recommended that the user run the RADX package described in Section 6.1.8, execute the data flow analysis capability of RADX (see Section 6.3) and correct the indicated anomalies.

As noted previously, the conventions concerning the RSL concepts are documented in Section 3.0; described below are the conventions for writing BETAs, GAMMAs, and TESTs.

7.1.1 Writing Beta and Gamma Models for Alphas

The required processing in an ALPHA is modeled for simulation in an executable description, either a BETA or GAMMA attribute. These executable descriptions are written as standard PASCAL procedures with the following alterations:

- The user does not provide a procedure heading -- this is done automatically by SIMGEN.
- Special statements (peculiar to REVS) are used to access FILEs -- these are translated by SIMGEN into legal PASCAL statements.

REVS processes the executable descriptions to produce standard PASCAL procedures for incorporation into the beta or gamma simulator.

All of the normal PASCAL programming features and facilities are available, except that all data that is not strictly local to a BETA or GAMMA must be declared via the RSL Data Segment (see Section 3.1). Communication between BETAs or GAMMAs of several ALPHAs during simulation is via the DATA described in the Data Segment. Specifically, a BETA or GAMMA communicates via the DATA INPUT to and OUTPUT from the ALPHA.

DATA specified in RSL are represented in simulations as variables of the same names and of the type assigned by the RSL attribute TYPE. Thus, reference to DATA from within a BETA or GAMMA is via the RSL name of the element consistent with the standard PASCAL reference conventions.

Accessing FILEs

FILEs consist of multiple instances of the DATA CONTAINED in the FILE. A series of statements which are unique to REVS and serve to augment the PASCAL language are used in BETAs and GAMMAs to distinguish between the multiple instances in FILEs and to create and destroy FILE entries. These file access statements, described below, are coded in the executable description as though they were PASCAL statements; they are translated into executable PASCAL code which accesses the FILE instances during simulation.

SELECT

The SELECT statement is used to make available to the BETA or GAMMA the contents of one record (instance) in a FILE. In BNF (see Appendix A), the syntax* of the statement is:

SELECT $\left\{\begin{matrix}\text{FIRST}\\ \text{NEXT}\end{matrix}\right\}_1^1$ RECORD FROM file-name

[SUCH THAT (<Boolean expression>)]

If FIRST is specified, the search operation begins with the first record in the FILE (a FILE is either ordered first-in-first-out or ORDERED by DATA CONTAINED in each record of the FILE). If NEXT is specified the search begins with the record immediately following the presently selected record.

If a SUCH THAT criteria is specified, the Boolean expression is evaluated using the DATA from the current instance. If the expression evaluates FALSE, the next instance is examined. If the expression evaluates TRUE, the predefined local DATA item RECORD_FOUND is set to a Boolean TRUE value, the contents of the record are made available to the BETA/GAMMA code, and the search terminates.

If no instance is found that meets the selection criteria, or if the FILE is empty, RECORD_FOUND is set to Boolean FALSE and the search is terminated.

The selected instance remains selected and thus its DATA values remain available until another selection is performed on the FILE or the net is terminated. The selection remains in effect even though the processing flow may pass from one ALPHA to another.

CREATE

The CREATE statement adds a new record to a FILE. The syntax of the statement is:

CREATE file-name RECORD

*The syntax of the special statements available for operating on FILEs in a BETA or GAMMA is summarized in Section 2 of Appendix G in both BNF and syntax diagram form.

When the record is created it is automatically selected and its data remain available until another CREATE or SELECT is performed on the FILE. During the creation of an instance all DATA items are initialized to their stipulated values from the INITIAL_VALUE attribute, or to a default value if none is given. After a CREATE, the BETAs/GAMMAs can assign new values to the data items via the usual PASCAL assignment statements.

DESTROY

The DESTROY statement removes the currently selected record from a FILE. The syntax of the statement is:

```
DESTROY file-name RECORD
```

After a DESTROY, the DATA values in the record are no longer available. The DESTROY performs a de-selection on the FILE; thus, the assignment of values to DATA CONTAINED in the FILE is meaningless until after the next selection.

FOR EACH

The FOR EACH statement is an iterative form of the SELECT statement. It allows a simple means of applying a common operation to multiple records in a FILE. The FOR EACH header specifies the FILE and the criteria (if any) to be used in selecting records. An ENDFOREACH signals the end of the range of code to be applied to each instance satisfying the evaluation criteria.

```
FOR EACH file-name RECORD [SUCH THAT (<Boolean expression>)] DO
     <PASCAL statement>
     ENDFOREACH
```

where PASCAL statement is defined to include a file access statement as a legal form.

Each instance is examined in turn, beginning with the first, and the logical expression evaluated using DATA from the instance. If the expression evaluates to true, the code encompassed by the DO and ENDFOREACH is executed using the instance DATA. If no instance meets the criteria, the FOR EACH is null and the embedded code is not executed. FOR EACH structures may be nested to ten deep.

To illustrate the use of the FOR EACH statement, assume that an ALPHA is to compute X_ERROR_SUM as the sum of all instances of DATA X_ERROR CONTAINED in FILE HISTORY. The following GAMMA would accomplish this summation.

```
    GAMMA:
        "BEGIN
         X_ERROR_SUM: = 0.0;
         FOR EACH HISTORY RECORD DO
            X_ERROR_SUM: = X_ERROR_SUM + X_ERROR
         ENDFOREACH;
         END;"
```

The same operation can be performed using SELECT statements as shown below:

```
    GAMMA:
        "BEGIN
         X_ERROR_SUM: = 0.0;
         SELECT FIRST RECORD FROM HISTORY;
         WHILE RECORD_FOUND DO BEGIN
            X_ERROR_SUM: = X_ERROR_SUM + X_ERROR;
            SELECT NEXT RECORD FROM HISTORY
            END
         END;"
```

<u>Simulating Alpha Entity and Message Operations</u>

The RSL relationships CREATES, DESTROYS, and SETS are between ALPHAs and ENTITY_CLASSes and ENTITY_TYPEs. They indicate that an ALPHA determines the existence of an instance in an ENTITY_CLASS (CREATE and DESTROY) and determines its specific ENTITY_TYPE (SETS). The relationship FORMS between an ALPHA and a MESSAGE indicates that the ALPHA designates that the MESSAGE will be PASSED through the appropriate OUTPUT_INTERFACE when the interface is encountered subsequently on the net.

When a simulator is generated, the code to perform these actions is automatically inserted in an ALPHA's executable description (BETA or GAMMA). CREATES and SETS are performed before any user specified code in the BETA or GAMMA -- all CREATES being performed first. DESTROYS and FORMS are performed immediately before exiting a BETA or GAMMA after any user specified code (if a BETA or GAMMA executable description is omitted from the ASSM, a BETA or GAMMA will be generated containing only these operations).

In response to a CREATES, a new entity in the ENTITY_CLASS will be established. All DATA items ASSOCIATED with the class will be initialized to their INITIAL_VALUEs or to default values if no INITIAL_VALUEs are specified. After a CREATES, the BETA/GAMMA or subsequent ones can assign new values to the DATA. The CREATES acts as an entity selection; once a new entity is created, it is selected until either 1) another ALPHA CREATES a new entity in the ENTITY_CLASS, 2) an ALPHA DESTROYS the newly created entity, 3) a SELECT or FOR EACH node whose subject is the ENTITY_CLASS or an ENTITY_TYPE which COMPOSES the class is traversed on the net, or 4) the net is terminated.

In response to a DESTROYS, the currently selected entity in the ENTITY_CLASS is destroyed. Since the DESTROYS is performed at the end of the BETA or GAMMA, DATA ASSOCIATED with the entity becomes unavailable upon exiting the ALPHA model. The DESTROYS acts as a de-selection on the class and its types.

In response to a SETS, the currently selected entity in the ENTITY_ CLASS COMPOSED of the ENTITY_TYPE will be assigned the specific ENTITY_TYPE. The selected entity may have been newly created, in which case it is assigned a particular ENTITY_TYPE and the DATA ASSOCIATED with the type will be assigned INITIAL_VALUES.

If the entity is not newly created and already has a type, the type will be changed and the DATA content of the entity will be changed as follows: DATA ASSOCIATED with only the previous type will be removed; DATA ASSOCIATED with only the new type will be assigned INITIAL_VALUES; and values of DATA ASSOCIATED with both types will be unaffected and remain available. In any case the values of DATA ASSOCIATED with the ENTITY_CLASS will remain available and unchanged. After a CREATES, a SETS must occur before either another CREATES occurs on the class, another entity is selected from the ENTITY_CLASS, or the R_NET terminates.

In response to a FORMs, conditions are established such that the MESSAGE will be PASSED through the OUTPUT_INTERFACE that PASSES the MESSAGE when the processing flow reaches the interface. DATA which MAKE the MESSAGE or which are CONTAINED in FILEs making the MESSAGE may be assigned values before or after the FORMs and until the processing flow reaches the OUTPUT_INTERFACE.

7.1.2 Writing Performance Requirements Tests

As described in the RSL Validation Segment (see Section 3.4), a PERFORMANCE_REQUIREMENT has an attribute TEST which defines the requirement as an executable test. The information available to a TEST is all DATA RECORDED by the VALIDATION_POINTs appearing on the VALIDATION_PATHs CONSTRAINED by the PERFORMANCE_REQUIREMENT. When generating a gamma simulator, REVS will also automatically generate a simulation post processor corresponding to the simulator; only the TESTs for PERFORMANCE_REQUIREMENTs CONSTRAINing those VALIDATION_PATHs which appear in the simulator are included in the post processor.

TESTs are written as pass/fail criteria. The executable TEST is translated by REVS into a Boolean valued PASCAL function whose name is the name of the PERFORMANCE_REQUIREMENT. The TEST is written as a standard PASCAL function with the following alterations:

- The user does not provide a function heading - this will be done automatically by SIMGEN.
- Special statements (peculiar to REVS) for accessing DATA are used - these are translated by SIMGEN into legal PASCAL statements.

RSL elements used in a TEST (i.e., DATA) are referenced directly by their RSL names. All RSL names available in a post processor are automatically declared by REVS; thus the only declarations which should appear in the TEST are those for variables, constants and types local to the TEST.

Conceptually, each time during simulation that the processing flow reaches a VALIDATION_POINT on a net, a RECORDING is generated consisting of all DATA RECORDED by the VALIDATION_POINT. A VALIDATION_POINT can RECORD a FILE; in which case DATA is extracted for each record in the FILE. The same DATA and FILEs may be RECORDED by many VALIDATION_POINTs. Thus, DATA referenced in a TEST must be uniquely identified by VALIDATION_POINT, by RECORDING, and by record in a FILE; a FILE must be uniquely identified by VALIDATION_POINT and RECORDING.

The approach used to establish uniqueness is analogous to that used in the remainder of RSL for DATA and FILEs ASSOCIATED with entities and for DATA CONTAINED in FILEs. Identification by RECORDING and by record in a FILE is performed by selection. Identification by VALIDATION_POINT is done through naming conventions. All RSL DATA and FILE names appearing in the TEST are prefixed by the name of the VALIDATION_POINT which RECORDED the DATA or FILE to be used. The two names are separated by a decimal point. (Thus, to refer to DATA (or FILE) B RECORDED by VALIDATION_POINT V1, the identifier V1.B is used in the TEST.)

The special operators for identifying a particular RECORDING are the RETRIEVE and FOR EACH. These operate identically to the SELECT and FOR EACH on FILEs written in BETAs and GAMMAs (see Section 7.1.1). The syntax of the RETRIEVE is shown below in BNF (see Appendix A).*

$$\text{RETRIEVE} \left\{\begin{matrix}\text{FIRST}\\ \text{NEXT}\end{matrix}\right\}_1^1 \text{RECORDING FOR validation-point-name}$$

[SUCH THAT (<Boolean expression>)]

To understand the operation of the RECORDING retrieval, the set of RECORDINGs generated by a VALIDATION_POINT can be thought of as an ordered assemblage with a pointer which may be moved. The ordering is from least to greatest on simulated time of generation of the RECORDING. The RETRIEVE statement first repositions the pointer: to the first recording for RETRIEVE FIRST; or to the one following the current pointer position for RETRIEVE NEXT. The condition is then evaluated using the DATA in the RECORDING designated by the pointer. If the expression evaluates to TRUE, the correct RECORDING has been found. Otherwise, the pointer is moved to the next instance and the process repeated. If the condition is omitted, the RECORDING will be RETRIEVED by positioning of the pointer only.

After a RETRIEVE operation, the predefined Boolean variable RECORDING_FOUND will have the value TRUE if a RECORDING which meets the retrieval was located; otherwise RECORDING_FOUND will have the value FALSE. The search does not go end-around; it is terminated and a value FALSE set when the last RECORDING is reached.

After a RECORDING has been RETRIEVED, all references to DATA and FILEs in the RECORDING are assumed to refer to that RECORDING. The RECORDING

*The syntax of the special statements available in TESTs is summarized in Section 3 of Appendix G in both BNF and syntax diagram form.

RETRIEVED remains available until the next RETRIEVE or FOR EACH on the same VALIDATION_POINT is encountered.

The syntax for the FOR EACH on VALIDATION_POINT RECORDINGs is shown below:

```
FOR EACH validation-point-name RECORDING
    [SUCH THAT (<Boolean expression>)] DO <PASCAL statement>
    ENDFOREACH
```

where PASCAL statement is defined to include a special TEST data access statement as a legal form.

The FOR EACH on RECORDINGs has the same meaning as the FOR EACH on FILEs in BETA/GAMMA descriptions (see Section 7.1.1); the executable statements encompassed by the DO and ENDFOREACH are executed for each RECORDING meeting the retrieval criteria. Again the RECORDINGs are examined in the order of their recording time. If the condition is omitted, the statements will be executed for all RECORDINGs generated by the VALIDATION_POINT.

Uniqueness by FILE is established by the special operators SELECT and FOR EACH. The syntax of these statements is shown below.

$$\text{SELECT} \begin{Bmatrix} \text{FIRST} \\ \text{NEXT} \end{Bmatrix}_1^1 \text{RECORD FROM validation-point-name.file-name}$$

```
    [SUCH THAT (<Boolean expression>)]
FOR EACH validation-point-name.file-name RECORD
    [SUCH THAT (<Boolean expression>)] DO <PASCAL statement>
    ENDFOREACH
```

These statements have the same interpretations as the SELECT and FOR EACH statements on FILEs in BETAs and GAMMAs (see Section 7.1.1). After a SELECT, the predefined Boolean variable RECORD_FOUND will have the value TRUE if a FILE record meeting the selection criteria was located; otherwise, it will have the value FALSE.

To illustrate the use of these statements, assume that the PERFORMANCE_ REQUIREMENT X_ERROR_LIMIT is to sum the X_ERROR RECORDED by the VALIDATION_ POINT V1 and to compare this to a maximum error. X_ERROR is contained in FILE HISTORY which is also RECORDED by V1. The following TEST would specify this requirement.

```
TEST:
    "CONST X_ERROR_MAX = 50.0;
     VAR X_ERROR_SUM: REAL;
     BEGIN
     X_ERROR_SUM: = 0;
     FOR EACH V1 RECORDING DO
        FOR EACH V1.HISTORY RECORD DO
            X_ERROR_SUM: = X_ERROR_SUM + V1.X_ERROR
            ENDFOREACH
        ENDFOREACH;
     X_ERROR_LIMIT: = (X_ERROR_SUM ≤ X_ERROR_MAX)
     END;"
```

Shown below is the equivalent TEST using RETRIEVE and SELECT statements instead of FOR EACH statements.

```
TEST:
    "CONST X_ERROR_MAX = 50.0;
     VAR X_ERROR_SUM: REAL;
     BEGIN
     X_ERROR_SUM: = 0;
     X_ERROR_LIMIT: = FALSE;
     RETRIEVE FIRST RECORDING FOR V1;
     WHILE RECORDING_FOUND DO
        BEGIN
        SELECT FIRST RECORD FROM V1.HISTORY;
        WHILE RECORD_FOUND DO
            BEGIN
            X_ERROR_SUM: = X_ERROR_SUM + V1.X_ERROR;
            SELECT NEXT RECORD FROM V1.HISTORY
            END;
        RETRIEVE NEXT RECORDING FOR V1
        END;
     IF X_ERROR_SUM > X_ERROR_MAX THEN X_ERROR_LIMIT: = FALSE
     END;"
```

7.2 INTERFACING A DRIVER

In order to generate a simulator, REVS must be provided with a driver. This driver is written externally to REVS in PASCAL and is provided to the Simulation Generation (SIMGEN) function on the Simulation Driver Definition File (SDF).

REVS provides a standard set of interface routines which the driver uses to communicate with the simulated data processing subsystem. The interface procedures and the conventions to be followed in writing a driver are described in the following sections.

7.2.1 Representing Subsystems

The data processing subsystem has interfaces to other components of the system. In RSL, INPUT_INTERFACEs and OUTPUT_INTERFACEs CONNECT the data processing subsystem to other SUBSYSTEMs. An INPUT_INTERFACE PASSES MESSAGEs to the data processing subsystem; an OUTPUT_INTERFACE PASSES MESSAGEs from the data processing subsystem.

Operation of a REVS generated simulator is based on the driver modeling these SUBSYSTEMs and accepting and generating the specified MESSAGEs. For each SUBSYSTEM specified in the requirements and used in the simulator, there must be a corresponding model in the driver. A SUBSYSTEM model must be capable of either:

1) accepting all MESSAGEs PASSED by the OUTPUT_INTERFACE which CONNECTS to the SUBSYSTEM;

2) generating all MESSAGEs PASSED by the INPUT_INTERFACE which CONNECTS to the SUBSYSTEM; or

3) performing both operations if the SUBSYSTEM CONNECTS to both an INPUT_INTERFACE and an OUTPUT_INTERFACE.

The SUBSYSTEM models are PASCAL procedures. The procedure names are the RSL names of the SUBSYSTEMs being modeled. These procedures have no calling parameters and must be provided on the SDF.

7.2.2 Sending Messages to the Simulated Data Processing Subsystem

During simulation, an R_NET is executed when a MESSAGE which is PASSED by an INPUT_INTERFACE becomes available at the interface. A driver SUBSYSTEM model causes invocation of an R_NET by posting MESSAGEs for a particular INPUT_INTERFACE using the following procedure call:

EEPSTMES (MESNAME, MESTIME)

where MESNAME is the RSL name of the MESSAGE to be posted at time MESTIME (real). EEPSTMES will post the MESSAGE for that INPUT_INTERFACE which PASSES the MESSAGE. The R_NET ENABLED by the INPUT_INTERFACE is scheduled for execution at MESTIME, the simulated time in seconds at which the MESSAGE is to be available at the interface. A SUBSYSTEM model may post many MESSAGES for an interface -- each will cause a separate invocation of the corresponding R_NET.

MESSAGEs may be MADE by FILEs. Before posting a MESSAGE MADE by a FILE, the driver must create the FILE records using the following procedure calls:

EEBLDREC (FILENAME) - A record for the FILE is created.

EESAVREC (FILENAME) - The newly created record is stored into the FILE structure.

The FILENAME is the RSL name of the FILE being created. The driver sets the values of DATA CONTAINED in the FILE record between calls to EEBLDREC and EESAVREC.

7.2.3 Obtaining Messages from the Simulated Data Processing Subsystem

When an OUTPUT_INTERFACE is traversed on a net, the MESSAGE FORMED for that interface will be posted for communication to the SUBSYSTEM CONNECTED to the OUTPUT_INTERFACE. Each posting of a MESSAGE will cause execution of the corresponding SUBSYSTEM model for processing of the MESSAGE.

In the simulator, interfaces are represented by lists containing the posted messages. A SUBSYSTEM model obtains access to messages on an interface list by operations analogous to the BETA/GAMMA SELECT FIRST and SELECT NEXT operations on FILEs. Only one message per interface list is available at a time. The PASCAL procedure calls described below are for use by the driver in accessing messages; in each case the input parameter INTNAME is the actual RSL name of the OUTPUT_INTERFACE.

EEFSTMES (INTNAME, FFLAG) - The first message on the interface list is selected and made available for reference. The found flag FFLAG is set to TRUE if a message is found or set to FALSE if a message is not found.

EENXTMES (INTNAME, DFLAG, FFLAG) - The next message on the interface list is selected and made available for reference. The found flag FFLAG is set to TRUE if a message is found or set to FALSE if a message is not found. If the input parameter destroy flag DFLAG is set TRUE, the previously selected message is destroyed before selecting the next message.

FILEs can MAKE MESSAGEs. Once a MESSAGE has been accessed, individual records in a FILE are accessed in a manner analogous to accessing the MESSAGE. The following procedure calls provide FILE access for the driver:

EEFSTREC (FILENAM, FFLAG) - The first record in the file named in the FILENAM parameter is selected and made available for reference. The found flag FFLAG is set to TRUE if a record is found or set to FALSE if a record is not found.

EENXTREC (FILENAME, FFLAG) - The next record in the named file is selected and made available for reference. The found flag FFLAG is set TRUE if a record is found or set to FALSE if a record is not found.

EEDSTREC (FILENAME) - The currently selected record in the file is destroyed. (A record must have been previously selected by a call to either EEFSTREC or EENXTRC.)

Although this discussion has been in terms of OUTPUT_INTERFACEs, these procedures can be used to access INPUT_INTERFACE lists if this is required in the driver.

7.2.4 Referencing Data

The MESSAGEs PASSED by an interface are MADE by DATA and by FILEs which CONTAIN DATA. In a simulator, only the lowest level of DATA (pertinent to the type of simulation) making a MESSAGE or CONTAINED in a FILE will be represented. A DATA item is represented by a PASCAL variable of the same name as the RSL element and of the type specified by the value of RSL attribute TYPE. Thus, the driver references DATA directly using the RSL name of the DATA. The variable and type declarations are generated automatically by REVS.

7.2.5 Scheduling Driver Events

During simulation, a SUBSYSTEM model is scheduled for execution whenever the CONNECTing OUTPUT_INTERFACE is traversed on a net; an R_NET is scheduled for execution in response to a SUBSYSTEM model posting a MESSAGE for the enabling INPUT_INTERFACE. In addition, the driver may execute SUBSYSTEM models independently of the R_NETs by scheduling the special driver procedure SSEXOG. SSEXOG may be scheduled to execute at any desired simulation time. This allows for the occurrence of internal driver events which are not in direct response to MESSAGEs from the simulated data processing subsystem.

The procedure available for scheduling driver exogenous events is:

```
PROCEDURE EECAUSE (SSEXOGSTR:EESTR;TIMEABS:REAL)
```

where SSEXOGSTR is a sixty character string containing the word SSEXOG as the first six characters and the remaining characters are blanks (EESTR=PACKED ARRAY [1..60] OF CHAR).

As with SUBSYSTEMS, a procedure named SSEXOG which has no calling parameters must be provided in the driver even if it is not used.

7.2.6 Obtaining Simulation Time

The current simulated time is available to the driver in the variable CLOCK_TIME; this is a pre-defined DATA of TYPE REAL and is therefore also available to the R_NETs. CLOCK_TIME should be treated as read only; it is reset from a master simulation clock just prior to execution of an R_NET, a SUBSYSTEM model or SSEXOG.

7.2.7 Initializing a Driver

At the beginning of a simulator execution the driver procedure SSSTARTUP will be invoked to allow for driver initialization. The procedure SSSTARTUP, which has no calling parameters, must be provided in the driver. SSSTARTUP performs any internal driver initialization required and, at a minimum, posts a message to the data processing subsystem to initiate the simulation. The user specified start time and end time are available to the driver as the values of the real data items EESTARTTIME and EESTOPTIME, respectively.

7.2.8 Organization of the Simulation Driver Definition File

The Simulation Driver Definition File (SDF) contains the source code for the driver components of a simulator. This file is constructed externally to REVS and is organized into four segments, separated by the character "$". Each segment of the SDF is inserted into its proper place in the simulator during execution of the Simulation Generation Function.

The format of the SDF is:

```
 Constant declarations
$
 Type declarations
$
 Variable declarations
$
 Model procedures
$
```

The first three segments contain any global constant, type, and variable declarations required by the driver. Any or all of these segments may be empty but the segment separators ("$") must be present. Also, the PASCAL keywords CONST, TYPE, and VAR must <u>not</u> be stated. The fourth segment contains the model procedures for the driver, and must contain a procedure SSSTARTUP and SSEXOG and a model procedure for each SUBSYSTEM required in the simulation. A model procedure must have the same name as the SUBSYSTEM name specified in RSL.

7.2.9 Naming Conventions

The RSL names for elements represented in the simulator (R_NETs, SUBNETs, ALPHAs, VALIDATION_POINTs, DATA, FILEs, ENTITY_TYPEs, ENTITY_CLASSes, MESSAGEs, SUBSYSTEMs, INPUT_INTERFACEs, and OUTPUT_INTERFACEs) are used directly as identifiers in the simulator. All of the simulation utilities are named with a two letter prefix of EE.

A driver should be written taking care not to duplicate any of these names. It is recommended that driver identifiers be defined using the two letter prefix SS.

7.3 GENERATING A SIMULATOR AND POST PROCESSOR (SIMGEN FUNCTION)

The generation of a simulator and post processor (for a gamma simulator) from the contents of the ASSM is performed automatically by invoking the REVS Simulation Generation (SIMGEN) function. The user exercises control over the scope and the type (beta or gamma) of simulator and may also provide an identifying name for the simulator. The SIMGEN commands to exercise these control are presented and described in the following subsection. The syntax of each of the commands is presented in BNF (see Appendix A); the complete syntax is summarized in Appendix G in both BNF and in syntax diagram form.

Several types of diagnostics may be issued after initiating SIMGEN: SIMGEN RCL diagnostics, data base analysis diagnostics (the RADX ANALYZE capability described in Section 6.3 is executed by SIMGEN), and SIMGEN translation diagnostics. The SIMGEN RCL and translation diagnostics are presented and explained in Appendix G; the data base analysis diagnostics are documented in Appendix F. In addition, since REVS checks only the special data access statements in the BETA/GAMMA and TEST attributes for correctness -- not the PASCAL statements, the PDL 2 or PASCAL compiler may issue error diagnostics when compiling the simulator and/or post processor. The error diagnostics issued by these compilers are explained in references 1 and 2, respectively.

Only one simulator and corresponding post processor may be generated, compiled and saved during an execution of REVS. See Section 9 for a description of the REVS macros related to the generation of simulators and post processors.

7.3.1 Defining the Scope of the Simulator

REVS does not require a complete set of requirements to be specified in the ASSM prior to simulation; only a portion of the requirements may be simulated (i.e., those pertaining to a collection of R_NETS). In commanding SIMGEN, the user specifies the particular R_NETs to be included in the simulator. Based on these R_NETs, SIMGEN will include only those RSL elements required to support simulation of the processing specified by these nets.

PRECEDING PAGE NOT FILMED BLANK

The user designates by either inclusion or exclusion the particular R_NETs to be simulated. In both cases the user supplies a list of the RSL names of the R_NETs. The inclusion statement has the following syntax:

$$\text{INCLUDE} \begin{bmatrix} \text{R\_NETS} \\ \text{R\_NET} \end{bmatrix} \left\{ \text{R-Net-name} \right\}_{1}^{n} .$$

Several inclusion statements may be used.

To specify the contents of the simulator by exclusion, a statement with the syntax

$$\text{EXCLUDE} \begin{bmatrix} \text{R\_NETS} \\ \text{R\_NET} \end{bmatrix} \left\{ \text{R-Net-name} \right\}_{1}^{n} .$$

is entered. Several exclusion statements can be used; the resulting simulator will contain all R_NETs specified in the ASSM except those appearing on the exclusion lists. A mixture of inclusion and exclusion statements is not allowed.

If SIMGEN is to use all R_NETs in the ASSM to generate a simulator the following command is entered:

INCLUDE ALL [R_NETS].

If this form of inclusion statement is used, the include-list form described above cannot also be used.

At least one INCLUDE or EXCLUDE statement must be provided to SIMGEN in order to generate a simulator.

7.3.2 Defining the Type of Simulator

In order for SIMGEN to generate a simulator, the user must designate the type, either beta or gamma, of simulation. The type declaration statement has the syntax

$$\begin{bmatrix} \text{SIMULATION} \\ \text{SIMULATOR} \end{bmatrix} \text{TYPE [IS]} \begin{Bmatrix} \text{BETA} \\ \text{GAMMA} \end{Bmatrix}_{1}^{1} .$$

At least one simulation type declaration must be provided; if multiple types are specified, the last entered will be used by SIMGEN.

7.3.3 Identifying the Simulator

The user may supply an identifier for a REVS generated simulator. The identifying name is entered in a SIMGEN statement using the following syntax:

$$\begin{bmatrix}\text{SIMULATION}\\ \text{SIMULATOR}\end{bmatrix}\left\{\begin{matrix}\text{ID}\\ \text{IDENT}\\ \text{IDENTIFICATION}\end{matrix}\right\}_{1}^{1}\ \text{[IS] identification-name.}$$

If multiple identification statements are provided to SIMGEN, only the last name entered will be used. The identification name will be used to label the output of the simulator and the output of the corresponding post processor (gamma simulation only).

7.4 EXECUTING A SIMULATOR (SIMXQT FUNCTION)

Simulators generated by REVS are PASCAL main programs which are executed as independent programs outside the direct control of REVS using the SIMRUN macro (see Section 9.0 for the REVS job control stream). User inputs to a simulator are provided, however, through the REVS Simulation Execution (SIMXQT) function.

These inputs must be provided through SIMXQT in order to execute a simulator. For any execution of REVS, only one set of inputs can be provided; if SIMXQT is executed more than once, only the last set of inputs will be routed to the simulator.

User inputs consist of simulation start and end times and a simulator run identification. These inputs are described below. The syntax is presented here in BNF (see Appendix A), and again in Section 1 of Appendix H in both BNF and syntax diagram form.

Diagnostic messages may be issued by both SIMXQT and by an executing simulator program; these are explained in Sections 2 and 3, respectively, of Appendix H.

7.4.1 Controlling Simulation Start and End Times

The user must, at a minimum, supply start and end times for simulation. The times are specified as real numbers in units of seconds. The syntax for these SIMXQT commands are:

$$\begin{bmatrix}\text{SIMULATION}\\ \text{SIMULATOR}\end{bmatrix} \text{ START [TIME] = real-number.}$$

$$\begin{bmatrix}\text{SIMULATION}\\ \text{SIMULATOR}\end{bmatrix} \text{ END [TIME] = real-number.}$$

Either start time or end time may be negative; however, end time must be strictly greater than start time. These two statements may be entered in either order but both must be present for simulator execution. If multiple start or multiple end times are provided, only the last times entered will be routed to the simulator.

In the simulator, the start time will be made available to the simulation driver in order to initiate simulation by posting messages to the data processing requirements model. During simulation, the RSL DATA item CLOCK_

PRECEDING PAGE NOT FILMED
BLANK

TIME reflects current simulated time. When the value of CLOCK_TIME equals or exceeds the user specified end time, the simulation is terminated.

7.4.2 Identifying the Execution of the Simulator

The user may supply an identification for any execution of a REVS generated simulator. The identifying name is entered in a SIMXQT control statement using the following syntax:

$$\begin{bmatrix}\text{SIMULATION}\\ \text{SIMULATOR}\end{bmatrix} \text{ RUN } \left\{\begin{matrix}\text{ID}\\ \text{IDENT}\\ \text{IDENTIFICATION}\end{matrix}\right\}_1^1 \text{ [IS] identification-name.}$$

If multiple identifications are provided to SIMXQT only the last name entered will be used. The identification name will be used to label the output of the simulator and the output of the corresponding post processor (gamma simulation only).

7.4.3 Simulator Output

The primary output of a simulator run consists of an RSL element execution trace and run time diagnostics. Each activation of an R_NET will be documented followed by the names of the nodes reflecting the path which was executed on that invocation of the net. Figure 7-2 shows a sample of the element trace.

Any run-time diagnostics will be output in the trace at the point at which an error was detected. The error diagnostics output during simulator execution are documented in Appendix H.

When executing a beta simulator, DATA RECORDED by VALIDATION_POINTs will be automatically output to a separate print file, a sample of which is shown in Figure 7-3. This file and the trace file will be printed automatically by the REVS job control stream documented in Section 9.

When executing a gamma simulator, a recording file is generated for DATA RECORDED by VALIDATION_POINTs. This file is processed by the post processor and, since it is not a standard print file, is not displayed when executing the simulator.

```
SIMULATION OUTPUT

SIMULATOR CREATED ON 06/27/77 AT 08:48:01 WITH ID TRACK_LOOP

RUN ID: EXAMPLE_FOR_USERS_MANUAL, DATE: 06/27/77, TIME: 09:51:50
SIMULATION START TIME =         0
SIMULATION END TIME =         5.000000E+00

                  R_NET CC_RESPONSE  BEGINS TO EXECUTE - TIME =          0
                        INPUT_INTERFACE CC_IN EXECUTES
                        ALPHA VALIDATE_HEADER EXECUTES
                        ALPHA ACKNOWLEDGE EXECUTES
                        OUTPUT_INTERFACE CC_OUT EXECUTES
                        ALPHA STARTER EXECUTES
                        VALIDATION_POINT STARTING_POINT EXECUTES
                        ALPHA ENGAGEMENT_INITIATION EXECUTES
                        EVENT SCHEDULE OCCURS
                        EVENT SUMMARIZE OCCURS
                  R_NET CC_RESPONSE  EXECUTION ENDS

                  R_NET SKED_R  BEGINS TO EXECUTE - TIME =          1.000000E-02
                        ALPHA INITIALIZE_SKED_R EXECUTES
                     FOR_EACH IMAGE_IN_TRACK BEGINS TO EXECUTE
                     FOR_EACH IMAGE_IN_TRACK ENDS EXECUTION
                     FOR_EACH CANDIDATE BEGINS TO EXECUTE
                     FOR_EACH CANDIDATE ENDS EXECUTION
                        EVENT XRB OCCURS
                  R_NET SKED_R  EXECUTION ENDS

                  R_NET XMIT_R  BEGINS TO EXECUTE - TIME =          1.000000E-02
                        ALPHA SELECT_COMMAND EXECUTES
                        EVENT SCHEDULE OCCURS
                  R_NET XMIT_R  EXECUTION ENDS
```

Figure 7-2 Sample Simulation Trace Output

```
VALIDATIØN DATA FØR RUN: EXAMPLE_FØR_USERS_MANUAL, DATE: 06/27/77 TIME: 09:51:50

 VALIDATIØN_PØINT:
STARTING_PØINT
 SIMULATIØN TIME =          0
 NØ DATA RECØRDED
 FILES RECØRDED
 FILE: WAVEFØRM_TABLE
 DATA RECØRDED
 RECØRD             1
 DATA: WF_CHARACTERISTICS
 VALUE:           1.000000E+00
 DATA: WF_NAME
 VALUE:             0
 RECØRD             2
 DATA: WF_CHARACTERISTICS
 VALUE:           2.000000E+00
 DATA: WF_NAME
 VALUE:             1
 RECØRD             3
 DATA: WF_CHARACTERISTICS
 VALUE:           3.000000E+00
 DATA: WF_NAME
 VALUE:             2

 VALIDATIØN_PØINT:
C2_IMAGE_HANDØVER
 SIMULATIØN TIME =          2.100000E-01
 DATA RECØRDED
 DATA: HØ_ID
 VALUE:             1
 NØ FILES RECØRDED
```

Figure 7-3 Sample Simulation Validation Point Output

7.5 EXECUTING A POST PROCESSOR (SIMDA FUNCTION)

Simulation post processors generated by REVS are PASCAL main programs which are executed as independent programs outside the direct control of REVS using the **TESTRUN macro** (see Section 9.0 for the REVS job control stream). User inputs to a post processor are provided, however, through the REVS Simulation Data Analysis (SIMDA) function. These inputs must be provided through SIMDA in order to execute a post processor. For any execution of REVS, only one set of post processor inputs can be provided; if SIMDA is executed more than once, only the last set of inputs will be routed to the post processor. In order for SIMDA to be used during a REVS execution, a post processor must have previously been generated by SIMGEN in this or a prior execution of REVS.

7.5.1 Controlling Performance Requirement Test Selection

User input to a post processor consists of the designation of the PERFORMANCE_REQUIREMENTs to be executed. The designated PERFORMANCE_REQUIREMENTs will be executed to evaluate the data recorded during execution of the corresponding analytic simulator.

The user may designate that either all or a subset of the PERFORMANCE_REQUIREMENTs be executed. To test the simulation results against all PERFORMANCE_REQUIREMENTs, the following command is entered into SIMDA:

$$\text{TEST ALL} \begin{bmatrix}\text{PERFORMANCE\_REQUIREMENTS}\\ \text{PERFORMANCE\_REQUIREMENT}\end{bmatrix}.$$

To select a subset of PERFORMANCE_REQUIREMENTs for execution, the user enters a command using the following syntax:

$$\text{TEST} \begin{bmatrix}\text{PERFORMANCE\_REQUIREMENTS}\\ \text{PERFORMANCE\_REQUIREMENT}\end{bmatrix} \left\{\text{performance-requirement-name}\right\}_1^n .$$

where the performance requirements names are the RSL names of the individual PERFORMANCE_REQUIREMENTs to be executed. The user can designate all the names in a single list or can enter this type of command several times with **different lists.**

If the list of requirements is quite long, the user may alternatively designate those PERFORMANCE_REQUIREMENTs which are not to be tested. Entry of the statement

$$\text{TEST ALL EXCEPT} \begin{bmatrix} \text{PERFORMANCE\_REQUIREMENTS} \\ \text{PERFORMANCE\_REQUIREMENT} \end{bmatrix}$$

$$\left\{\text{performance-requirement-name}\right\}_1^n .$$

will cause all PERFORMANCE_REQUIREMENTs in the post processor to be executed except those designated by names in the list. Clearly entry of more than one command of this form is not meaningful.

SIMDA will diagnose errors in the user input; the SIMDA diagnostic messages are explained in Appendix I. Also included in this appendix is the syntax of SIMDA RCL in both BNF and syntax diagram form.

7.5.2 Post Processor Output

The post processor output consists of the results (pass/fail) of execution of the PERFORMANCE_REQUIREMENT TESTs selected by the user and any post processor execution error messages. The test results output is labeled by simulator and simulation execution identification and by time and data of both the generation of the simulator and its execution. Shown below is sample output from a post processor.

```
SIMULATOR/TEST CREATED ON 06/27/77 AT 09:42:41 WITH ID TEST_PERFORMANCE_REQUIREMENTS.
RECORDING MADE ON 06/27/77 AT 09:54:59 WITH ID USERS_MANUAL_EXAMPLE.
PERFORMANCE REQUIREMENT EXAMPLE_PERFORMANCE_REQ PASSED.
```

Diagnostic messages output by a post processor are documented in Appendix I. Output is written to a standard print file which is printed automatically by the job control stream documented in Section 9.

8.0 EXTENDING THE LANGUAGE (RSLXTND FUNCTION)

The Requirements Statement Language may be extended or changed by the definition, modification, and deletion of element types, attributes, attribute legal values, and relationships. The remaining primitives of RSL, the Requirements Network STRUCTUREs and the validation PATHs, are only extensible in that the element types which can occur as element nodes on them are extensible.

Although RSL is extensible, the only parts of REVS which automatically adapt to these extensions are the RSL and RSLXTND functions and the RADX capabilities for subsetting and listing requirements (see Sections 6.1 and 6.2). The remaining portions of the REVS tools are dependent in varying degrees upon the existence of the element types, attributes, and relationships described in the RSL Alpha, Data, Validation, and Requirements Network Segments (see Sections 3.1 through 3.4). The only concepts within these segments which can be freely modified or deleted are the DATA attributes MAXIMUM_VALUE, MINIMUM_VALUE, RESOLUTION, and UNITS, and the VALIDATION_PATH attributes MAXIMUM_TIME, and MINIMUM_TIME. Extensions may, however, be freely made within the Management segment (see Section 3.5) with the exception that the element type SYNONYM and the relationship EQUATES have special significance and should never be altered.

To provide management control over changes in the language definition, a mechanism has been provided within the RSLXTND function which can be used to control extensions to the language. This control mechanism is the subject of Section 8.1. Other sections below detail and provide examples illustrating the extensions which may be performed on the language. The RSLXTND command syntax presented is expressed in the extended Backus-Naur Form (BNF) explained in Appendix A. The complete RSLXTND syntax from which these rules were extracted is presented in Appendix J, in both BNF and syntax diagram forms.

It should be noted that any RSL command is also a legal RSLXTND command. Also, the input and output specifications for the RSLXTND function are identical with those given for the RSL function in Section 5.1.

8.1 CONTROLLING LANGUAGE EXTENSIONS

The RSL extension function provides commands which may be used to establish and maintain control over extensions to the language. These extension controls are based on a two level system of permissions which can be entered into an ASSM and acquired by various users. The highest level of permission, control permission, denotes the permission to extend the language as well as the permission to establish and rescind permissions. Extension permission is subordinate to control permission and denotes the permission to extend the language but not the permission to establish or rescind permissions.

The use of controls over extension to the language is optional. If controls are not in use, every user of the RSL extension function (RSLXTND) has control permission and the ASSM is said to be uncontrolled. If controls are in use, the ASSM is said to be controlled and there will be at least one control permission established in the ASSM. All users will then have only the level of permission that they acquire by identifying themselves (see Section 8.1.3).

There are four types of extension control commands used to identify users, establish control permission, establish extension permission, and rescind permissions. These commands are discussed and illustrated in separate sections below. Each command syntax contains a name. These names are treated slightly different from the RSL names. They have only 58 significant characters instead of 60 and they are maintained separately from all other RSL names so that no conflict between them and RSL names can ever occur. The syntax for the extension control commands is:

```
<extension control command>::=
    IDENTIFICATION name.
  | EXTENSION_PERMISSION name.
  | CONTROL_PERMISSION name.
  | RESCIND PERMISSION name.
```

8.1.1 Establishing Control Permissions

An uncontrolled ASSM is changed into a controlled one by the establishment of a control permission. This is accomplished by stating the word CONTROL_PERMISSION followed by a name. For example, the following establishes a control permission:

PRECEDING PAGE NOT FILMED BLANK

```
CONTROL_PERMISSION CONTROL_ONE.
```

A user must have acquired control permission before he can establish a control permission. For an uncontrolled ASSM every user has control permission and any user may enter the first control permission, converting the ASSM to a controlled one. The user entering the first control permission retains control permission until he exits the RSLXTND function or identifies himself (see Section 8.1.3).

All control permissions in an ASSM are treated as equal. Any user with control permission can use that permission to establish further control and extension permissions (see Section 8.1.2), to rescind permissions (see Section 8.1.4), and to extend the language (see Section 8.2).

The complete syntax for establishing a control permission is:

```
<extension control command>::=
    CONTROL_PERMISSION name.
```

8.1.2 Establishing Extension Permissions

Once a control permission has been entered into the ASSM, an extension permission may be entered by stating the word EXTENSION_PERMISSION followed by a name. For example, assuming an uncontrolled ASSM to start, the following establishes a control permission MAGIC and an extension permission OPEN_SESAME.

```
CONTROL_PERMISSION MAGIC.
EXTENSION_PERMISSION OPEN_SESAME.
```

At least one control permission must exist in an ASSM before any extension permission may be entered. As long as at least one control permission exists in an ASSM, any number of extension permissions may be entered by anyone who has acquired control permission.

The complete syntax for establishing an extension permission is:

```
<extension control command>::=
    EXTENSION_PERMISSION name.
```

8.1.3 Identifying the User

If an ASSM is controlled, a user wishing to extend the language or to establish or rescind permissions must first acquire the appropriate level of permission. Permission is acquired by stating the word IDENTIFICATION followed by a name. This statement identifies the user and grants him the permission level which has been previously associated with the name specified. For example, assume that in a controlled ASSM control permissions CP_1 and CP_2 and extension permissions EP_1, EP_2, and EP_3 have been established. Either of the following statements would give the user extension permission, allowing him to extend the language (see Section 8.2):

```
IDENTIFICATION EP_1.
IDENTIFICATION EP_2.
IDENTIFICATION EP_3.
```

Either of the following statements would give the user control permission, allowing him to establish and rescind extension and control permissions as well as extend the language.

```
IDENTIFICATION CP_1.
IDENTIFICATION CP_2.
```

A user retains the permission level he has acquired until he either exits the RSLXTND function or re-identifies himself. The following example shows a user first acquiring extension permission and later acquiring no permission since there is no permission associated with NO_PER in our assumed ASSM. After the first statement, the user could extend the language; after the second, any attempted extensions are illegal.

```
IDENTIFICATION EP_2.
          .
          .  (language extensions legal)
          .
IDENTIFICATION NO_PER.
          .
          .  (language extensions illegal)
```

If an ASSM is uncontrolled, all users have implicit control permission and identification should not be used. If a user does identify himself and the ASSM is uncontrolled he will always acquire no permission.

The syntax for identifying a user is:

```
<extension control command>::=
    IDENTIFICATION name.
```

8.1.4 Rescinding Permissions

To rescind an existing permission, the words RESCIND PERMISSION, followed by a name are entered. To use this command, the user must have previously acquired control permission. For example, if control permission is associated with both KING and QUEEN and extension permission with COUNT and EARL, the following sequence will remove the permissions associated with QUEEN, COUNT, and EARL:

```
IDENTIFICATION KING.
RESCIND PERMISSION QUEEN.
RESCIND PERMISSION COUNT.
RESCIND PERMISSION EARL.
```

The permission associated with the name used to acquire control permission can also be rescinded but this rescission does not take effect until the next IDENTIFICATION command or the exiting of the RSLXTND function. Thus the following command sequence will remove all permissions in the above example, resulting in an uncontrolled ASSM.

```
IDENTIFICATION QUEEN.
RESCIND PERMISSION COUNT.
RESCIND PERMISSION KING.
RESCIND PERMISSION EARL.
RESCIND PERMISSION QUEEN.
```

Since control permission is necessary to rescind other permissions, the last remaining control permission in an ASSM cannot be rescinded as long as there is at least one extension permission outstanding. Thus the following sequence of commands is invalid since an extension permission for COUNT still exists:

```
IDENTIFICATION QUEEN.
RESCIND PERMISSION KING.
RESCIND PERMISSION EARL.
RESCIND PERMISSION QUEEN.
```

The syntax for rescinding a permission is:

```
<extension control command>::=
    RESCIND PERMISSION name.
```

8.2 DEFINING NEW RSL CONCEPTS

Using the RSLXTND functions, the user who has acquired extension or control permission may define new RSL concepts in the realm of new element types, new attributes, and new relationships. These RSL extension definitions are discussed in the following sections.

8.2.1 Defining a New Element Type

The syntax for defining a new element type is that of a new element type definition, which in its simplest form is:

```
[DEFINE] ELEMENT_TYPE element-type-name comment.
```

For example, the following defines a new element type named NEW_ONE:

```
DEFINE ELEMENT_TYPE NEW_ONE (*EXAMPLE*).
```

The word DEFINE may optionally precede the word ELEMENT_TYPE and its use is recommended. If DEFINE is not stated and the element type name is not in the ASSM, a new element type definition is assumed; however, if DEFINE is not stated and the element type name is in the ASSM, then the assumption is that the existing element type name is to be modified. Consequently, it is always safest to explicitly use the word DEFINE when the intent is to define a new element type. The following definition of NEW_ONE is thus the same as that above if NEW_ONE is not in the ASSM.

```
ELEMENT_TYPE NEW_ONE (*EXAMPLE*).
```

One additional sentence type, the structure applicability declaration, optionally preceded by the word INSERT, may be used to complete the new element type definition. This declaration has two forms to specify that the new element type may be used as an element node on a net STRUCTURE or on a validation PATH. The form of the structure applicability declaration is:

$$\text{STRUCTURE APPLICABILITY} \left\{\begin{matrix}\text{NET}\\ \text{PATH}\end{matrix}\right\}_1^1 .$$

As an example, the following defines three new element types. The first one, MY_TYPE, may be used as an element node on an R_NET or SUBNET STRUCTURE and on a PATH. The second one, YOUR_TYPE, is not usable on either a STRUCTURE or a PATH. The third one, THEIR_TYPE, is usable on a STRUCTURE but not on a PATH.

```
DEFINE ELEMENT_TYPE MY_TYPE (*MY OWN*).
    INSERT STRUCTURE APPLICABILITY NET.
    INSERT STRUCTURE APPLICABILITY PATH.
DEFINE ELEMENT_TYPE YOUR_TYPE (*NOT MINE*).
DEFINE ELEMENT_TYPE THEIR_TYPE (*NOT GOOD ON PATHS*).
    STRUCTURE APPLICABILITY NET.
```

The user who wishes to define new element types should be cautioned that the new element types defined will have no legal attributes, nor will they be legal subject or object element types for any relationships unless the desired attributes and relationships are modified to add the new element types to their applicable, subject, and object element type lists. The method of making these modifications is presented in Sections 8.3.2 and 8.3.3 for attributes and relationships, respectively.

The complete syntax for a new element type definition is as follows:

<new element type definition>::=
[DEFINE] ELEMENT_TYPE element-type-name comment.

$\left\{\text{[INSERT] <structure applicability declaration>}\right\}_0^n$

<structure applicability declaration>::=

$\text{STRUCTURE APPLICABILITY} \left\{\begin{matrix}\text{NET}\\ \text{PATH}\end{matrix}\right\}_1^1.$

8.2.2 Defining a New Attribute

The complete definition of a new attribute consists of three parts; a definition of a name and comment for the new attribute, a declaration of the element types to which the attribute applies, and a declaration of the values that the attribute may take on. The definition of a name and comment must precede the other two definition parts which have no imposed order. The form of the name and comment definition is one of the following:

```
ATTRIBUTE attribute-name comment.
DEFINE ATTRIBUTE attribute-name comment.
```

The other two parts of a new attribute definition are both forms of the attribute definition sentence, optionally preceded by the word INSERT. The applicable type declaration declares the element types to which the attribute applies. The two forms of the applicable type declaration are:

APPLICABLE ELEMENT_TYPE <element types>.

APPLICABLE <element types>.

There are three forms of the specification of <element types>:

ALL

ALL EXCEPT $\left\{\text{element-type-name}\right\}_1^n$

$\left\{\text{element-type-name}\right\}_1^n$

The first form means that elements of all currently defined element types except SYNONYM may take on values for the new attributes. The second form means that elements of all currently defined element types except SYNONYM and those **listed after** the word EXCEPT may take on values for the new attribute. The third form means that only elements of those specific element types listed may take on values for the new attribute.

The syntax for a new attribute definition allows any number of applicable type declarations. Multiple applicable type declarations are only meaningful, however, if all of them are in the third form, i.e., a list of element type names.

The values that the new attribute may assume are declared in one or more legal value declarations:

$$\text{VALUE} \left\{\begin{array}{l}\text{NUMERIC}\\ \text{TEXT}\\ \text{NAMED}\\ \text{value-name}\end{array}\right\}_1^1 \text{[comment].}$$

The value NUMERIC means that any signed or unsigned integer or real number as defined in PASCAL is a legal value for the new attribute (see Appendix B). The value TEXT means that any string of characters enclosed within double quotes is a legal value. The value NAMED means that any RSL name not used in another context in the ASSM is a legal value. The value name form means that the value name itself is a legal value.

Any number of legal value declarations may be included in a new attribute definition. Thus, it is possible to state that a new attribute may have as many legal values as desired,including any combination of NUMERIC, TEXT and either NAMED or any number of specific value names. Note that it is not meaningful to have legal values of NAMED and one or more specific value names for an attribute. These two legal value forms should <u>never</u> be used together.

The following example defines a new attribute NEW_AT which can apply to any element of type ALPHA or SUBNET and which can take on any NUMERIC or TEXT value:

```
DEFINE ATTRIBUTE NEW_AT (*COMMENT FOR NEW AT*).
    INSERT APPLICABLE ELEMENT_TYPE ALPHA, SUBNET.
    INSERT VALUE NUMERIC (*ANY NUMBER*).
    INSERT VALUE TEXT (*ANY TEXT STRING*).
```

As shown above, more than one legal value declaration can be given. The same is true for the applicable type declaration as long as none of them uses the ALL or ALL EXCEPT form. For example, the following is exactly equivalent to the attribute definition given above:

```
DEFINE ATTRIBUTE NEW_AT (*COMMENT FOR NEW_AT*).
    APPLICABLE ELEMENT_TYPE SUBNET.
    APPLICABLE ALPHA.
    VALUE TEXT (*ANY TEXT STRING*).
    VALUE NUMERIC (*ANY NUMBER*).
```

The following example defines a new attribute A_ONE with legal value VAL_ONE, VAL_TWO, and NUMERIC which may apply to elements of all currently defined element types except DATA (and except SYNONYM):

```
ATTRIBUTE A_ONE (*A_ONE*).
    VALUE VAL_ONE.
    VALUE NUMERIC.
    APPLICABLE ALL EXCEPT DATA.
    VALUE VAL_TWO.
```

This example illustrates that the applicable type and legal value declarations may occur in any order and may even be intermixed.

The complete syntax for a new attribute definition is:

<new attribute definition>::=

[DEFINE] ATTRIBUTE attribute-name comment.

$\left\{\text{[INSERT] <attribute definition sentence>}\right\}_0^n$

```
<attribute definition sentence>::=
      <applicable type declaration>
    | <legal value declaration>

<applicable type declaration>::=
     APPLICABLE [ELEMENT_TYPE] <element types>.

<element types>::=
     ALL
```

$$| \ \text{[ALL EXCEPT]} \left\{ \text{element-type-name} \right\}_1^n$$

```
<legal value declaration>::=
```

$$\text{VALUE} \left\{ \begin{matrix} \text{NUMERIC} \\ \text{TEXT} \\ \text{NAMED} \\ \text{value-name} \end{matrix} \right\}_1^1 \ \text{[comment].}$$

8.2.3 Defining a New Relationship

The definition of a new relationship consists of four parts; a definition of a name and comment for the relationship followed by relation definition sentences specifying the complementary relationship name, the element types which may be the subjects of the relationship, and the element types which may be the objects of the relationship. The form of the relationship name and comment definition is the following:

$$\text{[DEFINE]} \left\{ \begin{matrix} \text{RELATION} \\ \text{RELATIONSHIP} \end{matrix} \right\}_1^1 \ \text{relation-name}$$

```
[("relation-optional-word")] comment.
```

This part of the definition gives a name for the new relationship, called the primary relationship name, a comment, and may give a relation optional word which may be used in RSL commands following the use of the relation name to improve readability.

The three other parts of the new relationship definition are all forms of the relation definition sentence, optionally preceded by the word INSERT. The complementary relation declaration declares a name for the complementary relationship and may also specify a relation optional word intended to be used following the complementary relation name in RSL commands to improve readability.

COMPLEMENTARY $\left\{\begin{matrix}\text{RELATION}\\\text{RELATIONSHIP}\end{matrix}\right\}_1^1$ relation-name

[("relation-optional-word")].

Notice that no comment is allowed in the complementary relation declaration while one is required for the primary relationship name. Also, the primary and complementary relation names must always be distinct although their relation optional words have no such restriction. The guidelines followed in the core relationships for RSL (see Section 3) have been to always use the present tense, third person singular form of a verb for the primary relationship name, and to use the past tense, third person singular form of the same verb for the complementary relationship name. Relation optional words were used whenever their use would improve the readability and naturalness of RSL.

The element types which may be used as subjects of the new relationship are defined in a subject type declaration:

SUBJECT [ELEMENT_TYPE] <element types>.

There are three forms that the specification of the <element types> may take:

ALL

ALL EXCEPT $\left\{\text{element-type-name}\right\}_1^n$

$\left\{\text{element-type-name}\right\}_1^n$

The first form declares that elements of all currently defined element types, with the exception of SYNONYM, may be used as subjects of the new relationship. The second form declares that elements of all currently defined element types except those listed (and except SYNONYM) may be used as subjects of the new relationship. The third form lists specific element types which may be used as subjects of the new relationship.

The element types which may be used as objects of the new relationship are defined in an object type declaration:

OBJECT [ELEMENT_TYPE] <element types>.

As in the subject type declaration, there are three forms of the <element types> specification. The interpretation of these forms is identical except that references to subject element types should be read as references to object element types.

The syntax for a new relationship definition allows any number of relation definition sentences in any order. For a complete and valid new relationship definition the following rules should be adhered to:

- Exactly one complementary relation declaration should be specified.
- At least one subject type declaration and at least one object type declaration should be specified.
- Multiple subject or object type declarations should be specified only if all of them are in the form of a list of element type names.

The following example defines a new relationship PR_1 with relation optional word POW and complementary relationship CR_1 with relation optional word COW. The relationship's subject and object element types are all the currently defined element types except for SYNONYM.

```
DEFINE RELATIONSHIP PR_1 ("POW") (*NEW REL*).
    INSERT COMPLEMENTARY RELATIONSHIP CR_L ("COW").
    INSERT SUBJECT ELEMENT_TYPE ALL.
    INSERT OBJECT ELEMENT_TYPE ALL.
```

Assume that the only currently defined element types are ET_ONE, ET_TOW, and ET_THREE; then the following new relationship definition is exactly equivalent to the example above.

```
RELATION PR_1 ("POW") (*NEW REL*).
    OBJECT ET_ONE, ET_TWO.
    SUBJECT ALL.
    COMPLEMENTARY RELATION CR_1 ("COW").
    OBJECT ET_THREE.
```

This example illustrates that the three declaration parts may be in any order and that multiple subject or object type declarations may be used.

The complete syntax for defining a new relationship is:

<new relation definition>::=

[DEFINE] $\left\{\begin{matrix}\text{RELATION}\\\text{RELATIONSHIP}\end{matrix}\right\}_1^1$ relation-name

[("relation-optional-word")] comment.

$\left\{\text{[INSERT] <relation definition sentence>}\right\}_0^n$

<relation definition sentence>::=

<complementary relation declaration>

| <subject type declaration>

| <object type declaration>

<complementary relation declaration>::=

COMPLEMENTARY $\left\{\begin{matrix}\text{RELATION}\\\text{RELATIONSHIP}\end{matrix}\right\}_1^1$ relation-name

[("relation-optional-word")].

<subject type declaration>::=

SUBJECT [ELEMENT_TYPE] <element types>.

<object type declaration>::=

OBJECT [ELEMENT_TYPE] <element types>.

<element types>::=

ALL

| [ALL EXCEPT] $\left\{\text{element-type-name}\right\}_1^n$

8.3 MODIFYING RSL CONCEPTS

An existing element type, attribute, or relationship definition may be modified by using the RSLXTND commands to insert new information into or remove information from the definition. The user wishing to modify the standard RSL concepts should, however, be aware of the restrictions noted in Section 8.0. The following sections describe the various RSLXTND commands used to perform these modifications and provide examples illustrating their use.

8.3.1 Modifying an Element Type Definition

An element type definition may be modified by replacing the comment associated with the element type or by inserting or removing the declarations allowing its use as an element node on a net STRUCTURE or on a PATH. The element type modification must begin with a declaration of the element type which is to be modified:

```
[MODIFY] ELEMENT_TYPE element-type-name [comment].
```

The word MODIFY is optional but its use is suggested. If MODIFY is not used, the RSLXTND function assumes that this is a new element type definition if the element type name is not already in the ASSM. The comment is also optional; if a comment is provided it will replace the existing comment for the element type, otherwise the existing comment will be retained. The following two examples both modify an assumed element type ET_ASSUMED. The first one, however, does not change the definition of ET_ASSUMED while the second one gives ET_ASSUMED a new comment.

```
MODIFY ELEMENT_TYPE ET_ASSUMED.
MODIFY ELEMENT_TYPE ET_ASSUMED (*NEW COMMENT*).
```

The element type may be further modified by inserting or removing structure applicability declarations. The syntax for inserting a structure applicability declaration is:

$$\text{[INSERT] STRUCTURE APPLICABILITY} \left\{ \begin{matrix} \text{NET} \\ \text{PATH} \end{matrix} \right\}_1^1 .$$

The word INSERT is optional when inserting a structure applicability declaration.

The removal of a structure applicability declaration requires the use of the word REMOVE:

$$\text{REMOVE STRUCTURE APPLICABILITY} \left\{\begin{matrix}\text{NET}\\ \text{PATH}\end{matrix}\right\}_1^1 .$$

For example, assume that the following definition of element type TYPE_EX has been made:

```
DEFINE ELEMENT_TYPE TYPE_EX (*EX COMMENT*).
    INSERT STRUCTURE APPLICABILITY NET.
```

The following will then modify the definition of TYPE_EX so that it may be used as an element node on a PATH but not as an element node on a STRUCTURE.

```
MODIFY ELEMENT_TYPE TYPE_EX.
    INSERT STRUCTURE APPLICABILITY PATH.
    REMOVE STRUCTURE APPLICABILITY NET.
```

Note, however, that if one or more elements of the element type being modified are in use as element nodes on a STRUCTURE or PATH, the corresponding structure applicability declaration (NET or PATH) cannot be removed.

The complete syntax for an element type modification is:

<element type modification>::=
[MODIFY] ELEMENT_TYPE element-type-name [comment].

$$\left\{\begin{bmatrix}\text{INSERT}\\ \text{REMOVE}\end{bmatrix} \text{<structure applicability declaration>}\right\}_0^n$$

<structure applicability declaration>::=

$$\text{STRUCTURE APPLICABILITY} \left\{\begin{matrix}\text{NET}\\ \text{PATH}\end{matrix}\right\}_1^1 .$$

8.3.2 Modifying an Attribute Definition

The definition of an attribute consists of three parts; the declaration of a name and comment for the attribute, the declaration of the applicable element types for the attribute, and the declaration of the values that the attribute may assume. Any or all of these parts may be changed in order to modify the attribute definition.

Any modification of an attribute definition must begin with a declaration of the attribute name which is to be modified. The form of this declaration is:

[MODIFY] ATTRIBUTE attribute-name [comment].

If a comment is provided in this declaration it will replace the existing comment for the attribute, otherwise the existing comment will be retained.

The declaration of the attribute name and optional comment may be followed by any number of attribute declaration insertion or removal sentences. These sentences perform functions which are the subjects of the following sections. For convenience in reference, the first part of the syntax of the attribute modification is given below. The sections following will detail subordinate parts of the syntax as required.

<attribute modification>::=

[MODIFY] ATTRIBUTE attribute-name [comment].

$$\left\{\begin{array}{l}\text{[INSERT] <attribute definition sentence>}\\ \text{<applicable type declaration removal>}\\ \text{<legal value declaration removal>}\end{array}\right\}_0^n$$

<attribute definition sentence>::=

<applicable type declaration>

<legal value declaration>

8.3.2.1 Declaring New Applicable Types for an Attribute

New applicable types for an attribute are declared by giving an applicable type declaration, optionally preceded by the word INSERT.

[INSERT] APPLICABLE [ELEMENT_TYPE] <element types>.

The specification of <element types> is one of the forms:

ALL

ALL EXCEPT $\left\{\text{element-type-name}\right\}_1^n$

$\left\{\text{element-type-name}\right\}_1^n$

The first form specifies that all currently defined element types are to be added as applicable element types (except for SYNONYM). The second form specifies that all currently defined element types except those listed (and except SYNONYM) are to be added as applicable element types. The third form specifies specific element types which are to be added as applicable element types. If the attribute already has one or more applicable element types only the third form should be used as any duplicate applicable element types specified will result in errors.

An an example assume that element types ET_1, ET_2, ET_3, and ET_4 are currently defined but only ET_1 is an applicable element type for attribute AT_1. The following would establish ET_2, ET_3, and ET_4 as additional applicable element types for attribute AT_1:

```
MODIFY ATTRIBUTE AT_1.
    INSERT APPLICABLE ELEMENT_TYPE ET_2, ET_3, ET_4.
```

The same result would be accomplished with the following:

```
MODIFY ATTRIBUTE AT_1.
    APPLICABLE ET_4.
    APPLICABLE ET_2, ET_3.
```

The complete syntax for declaring new applicable types for an attribute is:

```
[INSERT] <applicable type declaration>

<applicable type declaration>::=
    APPLICABLE [ELEMENT_TYPE] <element types>.

<element types>::=
    ALL
```

$$| \text{ [ALL EXCEPT] } \left\{\text{element-type-name}\right\}_1^n$$

8.3.2.2 <u>Removing Existing Applicable Types for An Attribute</u>

One or more existing applicable types for an attribute may be removed by the following:

$$\text{REMOVE APPLICABLE [ELEMENT\_TYPE] } \left\{\begin{array}{l}\text{ALL}\\ \left\{\text{element-type-name}\right\}_1^n\end{array}\right\}_1^1 .$$

The form ALL means that all element types which are currently defined as applicable element types for the attribute will be removed as applicable element types. The result will be an attribute defined with no applicable element types. The second form lists specific element types which are to be removed as applicable element types for this attribute. An error will be detected if an element type listed is not an applicable element type for the attribute.

An element type may not be removed as an applicable element type for an attribute as long as any element of that element type has a value for the attribute. The attempted removal of such an applicable element type will be diagnosed as an error and rejected.

As an example, assume that attribute MY_AT has been defined with applicable element types MY_ET_1 and MY_ET_2. If no elements of type MY_ET_1 or MY_ET_2 exist with values for MY_AT then the following would remove both element types as applicable element types for MY_AT:

```
    MODIFY ATTRIBUTE MY_AT.
        REMOVE APPLICABLE ELEMENT_TYPE ALL.
```

The same result would be achieved by:

```
    MODIFY ATTRIBUTE MY_AT.
        REMOVE APPLICABLE MY_ET_2.
        REMOVE APPLICABLE MY_ET_1.
```

The complete syntax for removing an applicable type for an attribute is:

<applicable type declaration removal>::=

$$\text{REMOVE APPLICABLE [ELEMENT\_TYPE]} \left\{ \begin{matrix} \text{ALL} \\ \left\{\text{element-type-name}\right\}_1^n \end{matrix} \right\}_1^1 .$$

8.3.2.3 Declaring New Legal Values for an Attribute

New legal values for an attribute are declared by giving a legal value declaration, optionally preceded by the word INSERT.

$$\text{[INSERT] VALUE} \left\{ \begin{matrix} \text{NUMERIC} \\ \text{TEXT} \\ \text{NAMED} \\ \text{value-name} \end{matrix} \right\}_1^1 \text{[comment].}$$

The reader is referred to Section 8.2.2 for a discussion of the meanings of the words NUMERIC, TEXT, NAMED and value name.

Any number of legal values may be declared for an attribute, except that an attribute should <u>never</u> have legal values NAMED and one or more value names at the same time. For example, assume that attribute AT_ONE has been defined with legal values NUMERIC and NAME_ONE. Then the following would result in AT_ONE having legal values NUMERIC, TEXT, NAME_ONE, and NAME_TWO:

```
MODIFY ATTRIBUTE AT_ONE.
    INSERT VALUE NAME_TWO (*EXAMPLE*).
    INSERT VALUE TEXT.
```

The complete syntax for declaring new legal values for an attribute is:

[INSERT] <legal value declaration>

<legal value declaration>::=

$$\text{VALUE} \left\{ \begin{array}{l} \text{NUMERIC} \\ \text{TEXT} \\ \text{NAMED} \\ \text{value-name} \end{array} \right\}_1^1 \text{[comment].}$$

8.3.2.4 <u>Removing Existing Legal Values for an Attribute</u>

An existing legal value for an attribute may be removed by the following:

$$\text{REMOVE VALUE} \left\{ \begin{array}{l} \text{NUMERIC} \\ \text{TEXT} \\ \text{NAMED} \\ \text{value-name} \end{array} \right\}_1^1 .$$

Any number of attribute removal declarations may be given, each of which removes one existing legal value for the attribute being modified. No legal value for an attribute may, however, be removed as long as an element exists with a value of that form for the attribute.

As an example, assume that attribute AT_MINE is defined with legal values TEXT, NAME_1, NAME_2, and NUMERIC. Then the following would remove NUMERIC and NAME_1 as legal values for AT_MINE.

```
MODIFY ATTRIBUTE AT_MINE.
    REMOVE VALUE NAME_1.
    REMOVE VALUE NUMERIC.
```

The following would remove all legal values for AT_MINE and could only be accomplished if no element has any value for attribute AT_MINE.

```
MODIFY ATTRIBUTE AT_MINE.
          REMOVE VALUE TEXT.
          REMOVE VALUE NAME_1.
          REMOVE VALUE NUMERIC.
          REMOVE VALUE NAME_2.
```

The complete syntax for removing existing legal values for an attribute is:

<legal value declaration removal>::=

$$\text{REMOVE VALUE} \left\{ \begin{array}{l} \text{NUMERIC} \\ \text{TEXT} \\ \text{NAMED} \\ \text{value-name} \end{array} \right\}_1^1 .$$

8.3.3 Modifying a Relationship Definition

The definition of a relationship consists of four parts: (1) the declaration of a primary relationship name, comment, and optional relation optional word; (2) the declaration of a complementary relationship name and optional relation optional word; (3) the declaration of subject element types for the relationship; and (4) the declaration of object element types for the relationship. Any or all of these parts may be changed in order to modify the relationship definition.

Any modification of a relationship definition must begin with a declaration of the relationship name which is to be modified. The primary relationship name must be used in this declaration; the complementary relation name cannot be used. The form of the declaration is:

$$\text{[MODIFY]} \left\{ \begin{array}{l} \text{RELATION} \\ \text{RELATIONSHIP} \end{array} \right\}_1^1 \text{relation-name}$$

[("relation-optional-word")] [comment].

If a relation optional word is provided, it will be associated with the relation name, replacing the existing relation optional word if one existed. If no relation optional word is provided any existing relation optional word for the relation name will be retained. Also, if a comment is provided, it will replace the existing comment for the relationship, otherwise the existing comment will be retained.

The declaration of the relation name, optional relation optional word, and comment may be followed by any number of relation declaration insertion or removal sentences. These sentences perform the functions which are the subjects of the following sections.

For convenience in reference, the first part of the syntax of the relation modification is given below. The sections following will detail subordinate parts of the syntax as required:

```
<relation modification>::=

        [MODIFY] {RELATION     }1
                 {RELATIONSHIP }1  relation-name

            [("relation-optional-word")] [comment].

        ( [INSERT] <relation definition sentence>           )n
        { <complementary relation declaration removal>      }
        { <subject type declaration removal>                }
        ( <object type declaration removal>                 )0

<relation definition sentence>::=
        <complementary relation declaration>
      | <subject type declaration>
      | <object type declaration>
```

8.3.3.1 Declaring a New Complementary Relation Name for a Relationship

A new complementary relation name for a relationship may be declared by giving a complementary relation declaration optionally preceded by the word INSERT:

```
    [INSERT] COMPLEMENTARY {RELATION     }
                           {RELATIONSHIP }  relation-name
       [("relation-optional-word")].
```

As discussed in Section 8.2.3, the definition of a relationship requires that exactly one complementary relation name be declared for each relationship. Because of this, the only time that a new complementary relation name should be declared for an existing relationship is after the existing complementary relation name has been removed (see Section 8.3.3.2 below). Results are unpredictable if more than one complementary relation name exists for a relationship.

As an example, assume that relation REL exists and that no complementary relation exists for REL. The following will then define CREL with relation optional word COPT as the complementary relation name for REL.

```
     MODIFY RELATIONSHIP REL.
          INSERT COMPLEMENTARY RELATION CREL ("COPT").
```

With the same assumptions, the following would declare CREL_2 as the complementary relation name with no relation optional word.

```
     MODIFY RELATION REL.
          COMPLEMENTARY RELATIONSHIP CREL_2.
```

The complete syntax for declaring a new complementary relation name for a relation is:

[INSERT] COMPLEMENTARY $\left\{\begin{matrix}\text{RELATION}\\ \text{RELATIONSHIP}\end{matrix}\right\}_1^1$ relation-name

[("relation-optional-word")].

8.3.3.2 Removing a Complementary Relation Name for a Relationship

The complementary relation name for a relationship may be removed by giving a complementary relation declaration removal.

REMOVE COMPLEMENTARY $\left\{\begin{matrix}\text{RELATION}\\ \text{RELATIONSHIP}\end{matrix}\right\}_1^1$ relation-name

[("relation-optional-word")].

The complete definition of a relationship requires that a complementary relation name be defined, therefore the only time that a complementary relation name should be removed is when it is to be replaced by a new complementary relation name or in preparation for the removal of the relationship definition.

As an example of the removal of a complementary relation name assume that relation MY_REL with complementary relation name MY_COMP_REL is defined. Then the following will remove MY_COMP_REL.

```
     MODIFY RELATION MY_REL.
          REMOVE COMPLEMENTARY RELATION MY_COMP_REL.
```

If there is a relation optional word defined for the complementary relation name, that relation optional word will always be removed along with the complementary relation name, regardless of whether the complementary relation declaration removal included the relation optional word.

The complete syntax for the removal of a complementary relation is:

<complementary relation declaration removal>::=

REMOVE COMPLEMENTARY $\left\{\begin{matrix}\text{RELATION}\\ \text{RELATIONSHIP}\end{matrix}\right\}_1^1$ relation-name

[("relation-optional-word")].

8.3.3.3 Declaring New Subject Element Types for a Relationship

New subject element types for a relationship may be declared by giving a subject type declaration, optionally preceded by the word INSERT.

```
[INSERT] SUBJECT [ELEMENT_TYPE] <element types>.
```

The specification of <element types> is one of the forms:

ALL

ALL EXCEPT $\left\{\text{element-type-name}\right\}_1^n$

$\left\{\text{element-type-name}\right\}_1^n$

The first form specifies that all currently defined element types are to be added as subject element types (except for SYNONYM). The second form specifies that all currently defined element types except those listed (and except SYNONYM) are to be added as subject element types for the relation. The third form specifies specific element types which are to be added as subject element types. If the relationship already has one or more subject element types defined, only the third form should be used since any duplicate subject element types specified will result in the detection of errors.

As an example assume that element types TYPE_1, TYPE_2, and TYPE_3, are currently defined, with TYPE_3 defined as a subject element type for relation REL_1. The following would declare that TYPE_1 and TYPE_2 are also subject element types for relation REL_1:

```
MODIFY RELATIONSHIP REL_1.
    INSERT SUBJECT TYPE_2.
    SUBJECT ELEMENT_TYPE TYPE_1.
```

The complete syntax for declaring new subject element types for a relationship is:

```
[INSERT] <subject type declaration>

<subject type declaration>::=
    SUBJECT [ELEMENT_TYPE] <element types>.

<element types>::=
    ALL
  | [ALL EXCEPT {element-type-name}ⁿ₁
```

$\left| \text{[ALL EXCEPT } \left\{\text{element-type-name}\right\}_1^n \right.$

8.3.3.4 Removing Subject Element Types for a Relationship

One or more existing subject element types for a relation may be removed with a subject type declaration removal:

$$\text{REMOVE SUBJECT [ELEMENT\_TYPE]} \begin{bmatrix} \text{ALL} \\ \left\{\text{element-type-name}\right\}_1^n \end{bmatrix}.$$

The form ALL, or the lack of a specification of the element types to be retrieved, means that all element types which are currently defined as subject element types for the relation will be removed as subject element types. The result will be a relationship defined with no subject element types. The second form lists specific element types which are to be removed as subject element types for the relation. It is an error to list an element type which is not currently a subject element type for the relation.

A subject element type for a relation may only be removed if no element of that type is the subject of the relation.

As an example assume that relation PREL is defined with subject element types T_1, T_2, and T_3. The following will leave PREL with only T_2 as a subject element type:

```
MODIFY RELATION PREL.
    REMOVE SUBJECT ELEMENT_TYPE T_1, T_3.
```

The complete syntax for the removal of subject element types for a relation is:

$$\text{<subject type declaration removal>::=}$$
$$\text{REMOVE SUBJECT [ELEMENT\_TYPE]} \begin{bmatrix} \text{ALL} \\ \left\{\text{element-type-name}\right\}_1^n \end{bmatrix} .$$

8.3.3.5 <u>Declaring New Object Element Types for a Relationship</u>

New object element types for a relationship may be declared by giving an object type declaration, optionally preceded by the word INSERT.

```
[INSERT] OBJECT [ELEMENT_TYPE] <element types>.
```

The specification of <element types> is one of the forms:

$$\text{ALL}$$
$$\text{ALL EXCEPT} \left\{\text{element-type-name}\right\}_1^n$$
$$\left\{\text{element-type-name}\right\}_1^n$$

The first form specifies that all currently defined element types are to be added as object element types (except SYNONYM). The second form specifies that all currently defined element types except those listed (and except SYNONYM) are to be added as object element types. The third form lists specific element types which are to be added as object element types. If the relation being modified already has one or more object element types defined, only the third form should be used. Otherwise the duplicate declaration of object element types will result in the detection of errors.

As an example assume that relation REL_1 is defined with object element type ET_1. Also assume that element types ET_2 and ET_3 are defined. Then the following will result in ET_1, ET_2, and ET_3 being object element types for REL_1:

```
MODIFY RELATIONSHIP REL_1.
    INSERT OBJECT ET_3.
    INSERT OBJECT ELEMENT_TYPE ET_2.
```

The complete syntax for declaring new object element types for a relationship is:

```
[INSERT] <object type declaration>

<object type declaration>::=
    OBJECT [ELEMENT_TYPE] <element types>.

<element types>::=
    ALL
  | [ALL EXCEPT]{element-type-name}ⁿ₁
```

8.3.3.6 Removing Object Element Types for a Relationship

One or more object element types for a relation may be removed with an object type declaration removal:

$$\text{REMOVE OBJECT [ELEMENT\_TYPE]} \begin{bmatrix} \text{ALL} \\ \left\{\text{element-type-name}\right\}_1^n \end{bmatrix} .$$

The form ALL, or the omission of the specification of the object element types to be removed, means that all element types currently defined as object element types for the relation will be removed as object element types. The result will be a relationship with no object element types. The second form lists specific element types which are to be removed as object element types for the relation. An error will be detected if an element type is listed which is not an object element type for the relationship.

An element type may not be removed as an object element type for a relation if any element exists which is the object of that relation.

Assume that relationship REL_NAME is defined with object element types TYPE_1, TYPE_2, and TYPE_3. The following would remove all three as object element types.

```
MODIFY RELATION REL_NAME.
    REMOVE OBJECT ELEMENT_TYPE ALL.
```

The complete syntax for the removal of object element types for a relationship is:

$$\text{<object type declaration removal>::=}$$

$$\text{REMOVE OBJECT [ELEMENT\_TYPE]} \left[\begin{array}{l} \text{ALL} \\ \left\{ \text{element-type-name} \right\}_{1}^{n} \end{array} \right] .$$

8.4 DELETING RSL CONCEPTS

Any of the existing RSL element types, attributes, or relationships may be deleted using the RSLXTND function. The user should be aware, however, of the restrictions noted in Section 8.0. The following sections describe the RSLXTND commands used to perform these deletions and provide examples illustrating their use.

8.4.1 Deleting an Element Type

An element type may be deleted by specifying an element type deletion:

```
DELETE ELEMENT_TYPE element-type-name.
```

An element type may only be deleted if it is not in use. Any of the following constitutes a use of an element type which will prevent its deletion:

1. An element of the element type exists. The element must be deleted before the element type can be deleted (see Section 5.1.3).
2. The element type is an applicable element type for an attribute. The attribute must be modified to remove the element type as an applicable element type before the element type can be deleted (see Section 8.3.2.2).
3. The element type is a subject or object element type for a relationship. The relationship must be modified to remove the element type as a subject/object element type before the element type can be deleted (see Sections 8.3.3.4 and 8.3.3.6).

It is, however, not necessary to explicitly remove the structure applicability for an element type as this will be done automatically if all of the above conditions are met.

As an example, suppose that element type TYPE_OLD is defined but is not used, in the above sense. Then the following will delete TYPE_OLD, including its net or path structure applicability if any has been established:

```
DELETE ELEMENT_TYPE TYPE_OLD.
```

If TYPE_OLD had been, for example, an applicable element type for attribute AT_ONE but not otherwise in use, then the attribute AT_ONE would first have had to been modified to remove TYPE_OLD as an applicable element type before TYPE_OLD could be deleted. The following would accomplish this:

```
MODIFY ATTRIBUTE AT_ONE.
    REMOVE APPLICABLE ELEMENT_TYPE TYPE_OLD.
DELETE ELEMENT_TYPE TYPE_OLD.
```

The complete syntax for deleting an element type is:

```
<element type deletion>::=
    DELETE ELEMENT_TYPE element-type-name.
```

8.4.2 Deleting an Attribute

An attribute is deleted by an attribute deletion:

```
DELETE ATTRIBUTE attribute-name.
```

The only necessary condition for the deletion of an attribute is that no element may exist with a value for the attribute. If such an element exists, the attribute instance for the element must be removed or the element deleted before the attribute itself may be deleted. It is not, however, necessary to remove the applicable element types or legal values for the attribute. These will be automatically removed when the attribute is deleted.

As an example, assume that attribute AT_ONE is defined as follows:

```
DEFINE ATTRIBUTE AT_ONE (*EXAMPLE*).
    INSERT APPLICABLE ELEMENT_TYPE TYPE_1, TYPE_2.
    INSERT VALUE NUMERIC.
    INSERT VALUE TEXT (*ANY STRING OF CHARACTERS*).
```

If no element has a value for AT_ONE then the following would delete AT_ONE.

```
DELETE ATTRIBUTE AT_ONE.
```

The following sequence would also result in the deletion of AT_ONE.

```
MODIFY ATTRIBUTE AT_ONE (*THIS PART UNNECESSARY*).
    REMOVE APPLICABLE TYPE_1, TYPE_2.
    REMOVE VALUE TEXT.
    REMOVE VALUE NUMERIC.
DELETE ATTRIBUTE AT_ONE.
```

If the element EL_EXAMPLE had a value for AT_ONE the removal of that attribute instance would be required prior to the deletion of the attribute:

```
MODIFY TYPE_1 EL_EXAMPLE
     REMOVE AT_ONE.
DELETE ATTRIBUTE AT_ONE.
```

The complete syntax for the deletion of an attribute definition is:

```
<attribute deletion>::=
     DELETE ATTRIBUTE attribute-name.
```

8.4.3 Deleting a Relationship

A relationship definition is deleted by the statement:

$$\text{DELETE}\left\{\begin{matrix}\text{RELATION}\\ \text{RELATIONSHIP}\end{matrix}\right\}_1^1 \text{relation-name}$$

```
     [("relation-optional-word")].
```

The relation name used in this statement must be the primary relation name; a complementary relation name may not be used. As shown in the above syntax, the specification of the relation optional word is optional. Whether it is specified or not, the relation optional word for the relationship will be removed when the relationship is deleted.

Only one circumstance can prevent the deletion of a relationship. That circumstance is the existence of an instance of the relationship between elements in the ASSM. All instances of a relationship must be deleted before a relationship itself may be deleted. There is, however, no requirement to remove the complementary relation name or the subject or object element types before deleting a relationship. These, if they exist, will be automatically removed when the relationship is deleted.

Assume that relationship REL_EX is defined as follows:

```
DEFINE RELATION REL_EX ("OPT_WORD") (*JUST AN EXAMPLE*).
     INSERT COMPLEMENTARY RELATIONSHIP COMP_EX ("C_OPT_WORD").
     INSERT SUBJECT ALL.
     INSERT OBJECT ALL.
```

If no elements are related by REL_EX then the relationship may be deleted by:

```
DELETE RELATIONSHIP REL_EX.
```

Alternately, the relationship could be modified to explicitly remove the complementary relation name and subject and object element types before the relation was deleted. The result would be the same.

The complete syntax for deleting a relationship is:

<relation deletion>::=

$$\text{DELETE} \left\{ \begin{matrix} \text{RELATION} \\ \text{RELATIONSHIP} \end{matrix} \right\}_1^1 \text{relation-name}$$

[("relation-optional-word")].

9.0 REVS JOB CONTROL

REVS operates on the Control Data Corporation (CDC) 7600 and Texas Instruments Advanced Scientific Computer (ASC) at the Ballistic Missile Defense Advanced Technology Center (BMDATC) Advanced Research Center (ARC) in Huntsville, Alabama, and on the ASC at the Naval Research Laboratory (NRL) in Washington, D.C. The ARC installations of REVS operate either from card input (termed off-line mode) or interactively (termed on-line mode) using the Data Disc Color Graphics (ANAGRAPH) Display System Terminal. In the on-line mode, all REVS functions are available to the user and may be invoked in any order. In the off-line mode, all functions except RNETGEN may be utilized in any order. The NRL installation of REVS operates only in the off-line mode. An explanation of the job control stream required to execute REVS on each of these computers is documented in the following subsections.

9.1 TI-ASC JOB CONTROL

REVS is invoked as a program on the TI-ASC through the use of the Job Statement Language (JSL) of the operating system. To aid the user in executing REVS, the following JSL macros have been defined:

- REVSPREP to initialize all files
- REVSXQT to execute REVS
- SIMRUN to execute a REVS generated simulator
- TESTRUN to execute a REVS generated post processor.

Using these macros, the normal job setup is very simple, requiring a JOB card to identify the job, a LIMIT card to specify required disk and CPU resources, a MACASG card to obtain the REVS macros, the REVSPREP JSL macro call, and the REVSXQT macro call followed by REVS RCL and RSL. A job set-up includes macro calls to SIMRUN and TESTRUN, if, respectively, a simulator and/or simulation post processor are to be executed. Any files to be referenced with an ADDFILE statement within REVS must be provided by the user. The last card of the job is the // EOJ card.

The JOB and LIMIT cards are documented in the TI-ASC JSL reference manual [3]. The MACASG statement is installation dependent as follows:

- At the BMDATC Advanced Research Center (ARC):
  `// MACASG M,USERCAT/TRW/REVS/MACROS`
- At the Naval Research Laboratory (NRL):
  `// MACASG M,AFFIL/IND/TRW/BERGP1/REVS/MACROS`

Each of the REVS macros is described in the following subsections; Section 9.1.5 contains sample REVS job decks.

9.1.1 REVSPREP Macro

The REVSPREP macro provides for the acquisition of all necessary files and environmental conditioning for use of REVS on the TI-ASC. The parameters for this macro are defined in Table 9.1. Defaults for the parameters are listed in the table; the current parameter defaults are also documented in the macro expansion in the job activity file when the macro is used.

PRECEDING PAGE NOT FILMED
BLANK

Table 9.1 REVSPREP Macro Parameters

PARAMETER	FUNCTION	DEFAULTS
REVSVERS= version	Specifies the disk version of REVS	+0
REVSEFID= tape number	Specifies tape number of REVS baseline to be used	Disk version to be used
ASSMEFID= tape number	Specifies tape number of ASSM to be staged to disk	Disk version to be used
ASSMFILE= file number	Specifies file number of ASSM on tape if ASSMEFID specified	1
ASSMBAND= bands	Specifies maximum band size for ASSM (4 pages/band)	100
ASSMVERS= version	Specifies the disk version of baseline ASSM	+0
SIMEFID= tape number	Specifies tape number of simulator to be used	No simulator loaded from tape
SIMGEN= {YES / NO}	Specifies whether SIMGEN is to be executed in this job (applies to NRL only)	NO

The REVSPREP must be used once and only once per job and must precede calls to any other REVS macros. Any number of REVS executions can be specified following the REVSPREP card. Normally, only one execution is required; with few exceptions (see Section 7.0), any number of REVS functions can be executed in any order within one step.

9.1.2 REVSXQT Macro

The REVSXQT macro invokes execution of REVS and provides all control necessary to acquire and dispose files needed during the REVS step. Table 9.2 contains an explanation of the macro parameters and their default values.

9.1.3 SIMRUN Macro

This JSL macro executes a simulator generated by the SIMGEN function of REVS. The simulator may have been generated in a previous REVSXQT step, or loaded from tape by the REVSPREP macro, either of which will supply the associated files needed. Execution of the SIMXQT function in a previous REVSXQT step is required to supply necessary simulator input. The simulator may be saved after execution if recording data has been collected for a simulation post processor which is not to be executed in this job. The SIMRUN macro parameters are defined in Table 9.3. Parameters on the PASCAL macro PXQT are available in addition to those shown in the table.

9.1.4 TESTRUN Macro

This JSL macro executes a simulation post processor generated by the SIMGEN function of REVS. The post processor may have been generated in a previous REVSXQT step, or loaded from tape by the REVSPREP macro. The recording data base used by the post processor is generated by execution of a gamma simulator, and is saved on tape with a simulator (a null data base is generated if the simulator is saved prior to execution in a SIMRUN step). Execution of the SIMDA function in a previous REVSXQT step is required to supply necessary post processor control input. Table 9.4 contains an explanation of the TESTRUN macro parameters and their default values. In addition to those shown in the table, parameters of the PASCAL macro PXQT are available.

Table 9.2 REVSXQT Macro Parameters

PARAMETER	FUNCTION	DEFAULTS
GO= access name	Specifies program file to be executed	REVSABS
REVSIN= access name	Specifies the access name of the REVS input file	Unnamed file of cards immediately following macro
ADDMEM= memory size	Specifies additional memory required for file buffers (SIMGEN requires 28K)	12K
STKSIZE= words	Specifies number of words to be allocated to the stack	10000
HEAPSIZE= words	Specifies number of words to be allocated for heap	10000
OUTBAND= band sizes	Specifies band allocation for REVS.OUT file	1/50/1
DMPBAND= band sizes	Specifies band allocation for REVS.DMP debug file	1/10/1
OPT= options	Specifies step options	(I)
CPTIME= seconds	Specifies step time limit in seconds	600
CALCOMP= {YES / NO / FOSYS / tape number}	Specifies disposition of CALCOMP plot output. NO means suppress tape unconditionally. YES means produce tape only if plots are actually generated. Plotting requires tape label for file FT04F001 at the ARC. At NRL, plotting is performed automatically. At NRL, FOSYS indicates that the online plotter is to be used. At NRL, a tape number can be specified if the plot tape is to be saved by the user.	YES
ASSMSAVE= {YES / NO}	Specifies disposition of ASSM after execution. YES means save on tape (requires label for file FT02F001). NO means leave on disk for subsequent exection, or discard.	NO

Table 9.2 REVSXQT Macro Parameters (Continued)

PARAMETER	FUNCTION	DEFAULTS
REVSPNCH= {YES / NO / access name}	Specifies disposition of punch output. YES means punch cards if RADX has been instructed to punch. NO means suppress punching even if RADX punch statements are executed. Specifying an access name causes the punch file to be retained as a job local file with the specified access name for subsequent user disposition.	YES
SIMGEN= {YES / NO}	Specifies whether SIMGEN will be used in this step. YES means SIMGEN may be executed. NO means SIMGEN will not be executed and therefore no capability to build a simulator or post processor is to be provided.	NO
SIMSAVE= {YES / NO}	Specifies whether to save a generated simulator on tape. YES means that if a simulator was generated it will be saved on tape with all adjunct files required, which includes the ASSM, SDF file and simulation post processor, if any. This tape can be subsequently reloaded by the REVSPREP macro for execution of a simulator or simulation post processor. NO means don't save on tape.	NO
REVSLOG= {KEEP / *}	Specifies whether to suppress printing of the REVSLOG output. KEEP means do not print but leave file for subsequent user disposition (for use by interactive users at NRL), * means print it.	*
REVSOUT= {KEEP / *}	Specifies whether to suppress printing of REVSOUT output. KEEP means do not print but leave file for subsequent user disposition (for use by interactive users at NRL), * means print it.	*

Table 9.3 SIMRUN Macro Parameters

PARAMETER	FUNCTION	DEFAULTS
STKSIZE= words	Specifies number of words to be allocated to the run time stack.	5000
HEAPSIZE= words	Specifies number of words to be allocated to the run time heap.	5000
ADDMEM= memory size	Specifies additional memory required for file buffers.	15000
CPTIME= seconds	Specifies step time limit in seconds	240
SIMSAVE= $\left\{\begin{matrix}\text{YES}\\\text{NO}\end{matrix}\right\}$	Specifies whether to save simulator, simulation post processor, recording data base and associated files on tape. Appropriate only for gamma simulations.	NO

Table 9.4 TESTRUN Macro Parameters

PARAMETER	FUNCTION	DEFAULTS
STKSIZE= words	Specifies number of words to be allocated to the run time stack	5000
HEAPSIZE= words	Specifies number of words to be allocated to the run time heap	5000
ADDMEM= memory size	Specifies additional memory required for file buffers	15000
CPTIME= seconds	Specifies step time limit in seconds	240

9.1.5 <u>Sample REVS TI-ASC Job Decks</u>

Figures 9-1 and 9-2 are excerpts from sample REVS TI-ASC job decks. Figure 9-1 shows a deck setup to execute REVS using three ADDFILEs. The other sample, Figure 9-2, is a deck setup to generate and execute a simulator.

```
// JOB BIN-14-ERRFIND,PTLAZ,SMITH
// LIMIT BAND=200,MIN=60
// MACASG MACROS,USERCAT/TRW/REVS/MACROS
//   REVSPREP ASSMEFID=6304
// REVSXQT CPTIME=3000,HEAPSIZE=30000
RADX.
ADDFILE HIER.
ADDFILE LISTSET.
ADDFILE ERRORS.
STOP.
// START ACNM=HIER
HIER SUBSYS_TO_DATA = SUBSYSTEM CONNECTED INPUT_INTERFACE,
                      INPUT_INTERFACE PASSES MESSAGE,
                      MESSAGE MADE BY FILE,
                      MESSAGE MADE BY DATA,
                      FILE CONTAINS DATA,
                      DATA INCLUDES DATA.
                               •
                               •
                               •
HIER ET_ASSOCIATES = ENTITY_TYPE ASSOCIATES FILE,
                     ENTITY_TYPE ASSOCIATES DATA,
                     FILE CONTAINS DATA,
                     DATA INCLUDES DATA.
// STOP
// START ACNM=LISTSET
SET STD_DATA = FOUND,RECORD_FOUND,CLOCK_TIME.
                               •
                               •
                               •
SET OF_ENTITY_RELATED_DATA = ALL BY HIER ECLASS.
SET OF_MESSAGE_DATA = ALL WITH HIER MSG.
SET OF_FILE_NAMES = FILE.
SET OF_DATA_CONTAINED_IN_FILES = ALL BY HIER FILES.
SET OF_FULLY_CROSS_REFERENCED_REQUIREMENTS_INFORMATION = ALL.
SET DATA_WITH_ENUM = SPECIFIED_DATA WITH TYPE  = ENUMERATION.
                               •
                               •
                               •
// STOP
// START ACNM=ERRORS
APPEND ALL NONE.
LIST OF_DATA_WITH_NO_SOURCE_OR_SINK.
LIST OF_DATA_WITH_SOURCE_BUT_NO_SINK.
                               •
                               •
                               •
LIST OF_EMPTY_NETWORKS.
LIST OF_UNUSED_NETWORK_ELEMENTS.
APPEND ALL ALL.
// STOP
// EOJ
```

Figure 9-1 REVS TI-ASC Job Deck - Sample 1

```
// JOB JOBNAME,JOBNO,USERNAME
// LIMIT BAND=200,MIN=20
// MACASG MACROS,USERCAT/TRW/REVS/MACROS
// REVSPREP ASSMVERS=+2
// REVSXOT ADDMEM=28K,CPTIME=1200,HEAPSIZE=20000,SIMGEN=YES
SIMGEN.
SIMULATION TYPE BETA.
INCLUDE ALL R_NETS.
SIMULATOR IDENT IS TRACK_LOOP.
SIMXQT.
START=0.0.
END=5.0.
RUN ID IS EXAMPLE_FOR_USERS_MANUAL.
STOP.
// SIMRUN OUTBAND=8/24/3
// TESTRUN
// START ACNM=SDF
  (*SETS CONSTANT DECLARATIONS*)
SSCCIN  ='CC_IN
SSRDRIN ='RADAR_IN
SSRCLKIN='RADAR_CLOCK_IN
$
  (*SETS TYPE DECLARATIONS*)
  SSTRING = PACKED ARRAY [1..28] OF CHAR;
$
  (*SETS VARIABLE DECLARATIONS*)
  SSFOUND  : BOOLEAN;
  SSNUMR : REAL;
  SSEED  : REAL;
  SST3COUNT : INTEGER;
    SST3CNTR : INTEGER;
  SSOUTPUT : TEXT;
$
  (*SETS PROCEDURES*)
PROCEDURE SSEXOG;
BEGIN  (* NULL EXOGENOUS EVENT PROCEDURE *)
END;
PROCEDURE SSSTARTUP;

                              •
                              •
                              •

  END;
$
// STOP
// EOJ
```

Figure 9-2 REVS TI-ASC Job Deck - Sample 2

9.2 CDC 7600 JOB CONTROL

REVS is invoked as a series of programs on the CDC 7600 through the use of the job control statements of the operating system. The following programs emulating the JSL macros defined for REVS on the TI-ASC have been defined:

- REVSPRE to initialize all files
- REVSXQT to execute REVS
- SIMRUN to execute a REVS generated simulator
- TESTRUN to execute a REVS generated post processor
- SIMSAVE to save a REVS generated simulator and post processor
- SIMLOAD to reload a REVS generated simulator and post processor.

Using these programs, the normal job setup can be very simple, requiring a job card to identify the job and acquire necessary resources, ATTACH and LIBRARY cards to obtain the library of REVS programs, a REVSPRE card, an EXIT(U) card, and a REVSXQT card followed by a end-of-section (7/8/9) card and REVS RCL and RSL. The last card of the job deck is an end-of-information (6/7/8/9) card.

A job setup may include additional job control statements, as necessary, to acquire and save ASSM files and to provide auxiliary input files, such as files referenced with an ADDFILE statement within REVS. The job setup may also include SIMRUN and TESTRUN cards if, respectively, a simulator or simulation post processor are to be executed. A SIMSAVE card may be included to prepare simulation related files for storage on tape or disk. These files may then be made available in a subsequent job by a SIMLOAD card.

The REVS programs are described in the following subsections. The required ATTACH and LIBRARY cards are as follows:

```
ATTACH,REVSLIB,TRWREVS7600SYSTEM,ID=PTLREVS.
LIBRARY,REVSLIB.
```

Additional job control statements are documented in the CDC SCOPE 2.1 reference manual [4]. Sample REVS job decks are illustrated in Section 9.2.7.

9.2.1 REVSPRE Program

The REVSPRE program provides for the acquisition of necessary files and environmental conditioning for use of REVS on the CDC 7600. This includes the acquisition of an ASSM containing the predefined element types, attributes, relationships, and elements described in Section 3. This ASSM may be replaced by an existing ASSM saved as a catalogued file by placing the following two control cards after the REVSPRE card:

```
RETURN,TAPE2.
GETPF,TAPE2,PERMFILENAME,ID=YOURID.
```

If the ASSM is on tape rather than on a catalogued file then the GETPF should be replaced by an appropriate STAGE or REQUEST card as described in [4], specifying TAPE2 as the local file name.

The REVSPRE must be used once and only once per job and must precede calls to any other REVS programs. Any number of REVS executions can be specified following the REVSPRE card. Normally, only one execution is required; with few exceptions (see Section 7.0), any number of REVS functions can be executed in any order within one step. The job control statement required to invoke the REVSPRE program is simply the word REVSPRE followed by a period; there are no parameters defined.

9.2.2 REVSXQT Program

The REVSXQT program invokes execution of REVS and controls the acquisition and disposal of files needed during the REVS step. To insure complete processing by REVS, an EXIT(U) should always immediately precede a REVSXQT card. REVSXQT provides six positional parameters representing file names as follows:

```
REVSXQT(FILE01,FILE02,FILE03,FILE04,FILE05,FILE06)
```

The parameters and their default values are:

File	Default	Interpretation
FILE01	INPUT	Standard REVS input file (REVS.IN)
FILE02	OUTPUT	Standard REVS log file (REVS.LOG)
FILE03	OUTPUT	Standard REVS output file (REVS.OUT)
FILE04	OUTPUT	DBCS TAPE6 error file (DBCS.ERR)
FILE05	OUTPUT	SIMGEN debug output file (REVS.DMP)
FILE06	PUNCH	Standard REVS punch file (REVS.PUN)

All files with the default name of OUTPUT will be automatically printed by REVS unless the file names are overridden on the REVSXQT card. The standard REVS punch file will also be automatically punched. If any of these file names are overridden on the REVSXQT call, the user assumes responsibility for their proper disposition.

The REVSXQT program provides for the automatic compilation of a simulator and post processor if the SIMGEN function is invoked. No special user action is required to accomplish this. REVSXQT also provides for the automatic post-staging of any plot tape if plotting was requested through RNETGEN or RADX. The local file name for the plot tape is TAPE 4 and a tape label must be submitted by the user.

9.2.3 SIMRUN Program

The SIMRUN program executes a simulator generated by the SIMGEN function of REVS. The simulator may have been generated in a previous REVSXQT step, or loaded from a tape or disk file by the SIMLOAD program (see Section 9.2.6), either of which will supply the associated files needed. Execution of the SIMXQT function in a previous REVSXQT step is required to supply necessary simulator input. The SIMRUN program is invoked by a control card specifying the word SIMRUN followed by a period; there are no parameters defined. Also, to insure complete processing by REVS, an EXIT(U) should always immediately precede a SIMRUN card.

9.2.4 TESTRUN Program

The TESTRUN program executes a simulation post processor generated by the SIMGEN function of REVS. The post processor may have been generated in a previous REVSXQT step, or loaded from a tape or disk file by the SIMLOAD program (see Section 9.2.6). The recording data base used by the post processor is generated by execution of a gamma simulator, and is saved along with a simulator by the SIMSAVE program (see Section 9.2.5). (A null data base is generated if the simulator is saved prior to execution in a SIMRUN step.) Execution of the SIMDA function in a previous REVSXQT step is required to supply necessary post processor control input. The TESTRUN program is invoked by a control card specifying the word TESTRUN followed by a period; there are no parameters defined.

9.2.5 SIMSAVE Program

The SIMSAVE program consolidates all files necessary to execute a simulator and post processor generated by the SIMGEN function of REVS.

These files are combined on a local file named SIMFILE. This file can then be saved on disk or tape by using the appropriate CATALOG, REQUEST, or STAGE cards as described in [4]. The files combined on SIMFILE include the load modules for the simulator and post processor programs as well as the other files necessary for their execution. The recording data base generated by execution of a gamma simulator is included (a null data base is generated if the simulator is saved prior to execution in a SIMRUN step). The SIMSAVE program is invoked by a control card specifying the word SIMSAVE followed by a period; no optional parameters are defined.

9.2.6 SIMLOAD Program

The SIMLOAD program reconstructs REVS simulator and post processor files previously saved by the SIMSAVE program. The SIMLOAD program assumes that these files are available on a local file named SIMFILE; the user is responsible for supplying the necessary ATTACH, REQUEST, or STAGE card as described in [4] to obtain this local file. After the SIMLOAD program runs, the loaded simulator and post processor may be executed by the SIMRUN and TESTRUN programs after the required SIMXQT and SIMDA inputs are supplied in a REVSXQT step.

9.2.7 Sample REVS CDC 7600 Job Decks

This section illustrates a number of sample deck setups to execute REVS on the CDC 7600. Some familiarity with the CDC SCOPE operating system job control statements as defined in [4] is assumed.

9.2.7.1 Nominal Execution

The deck setup below illustrates a minimal job starting with the predefined ASSM and saving no resultant ASSM.

```
JOBNO,STMFZ,T77.                                          YOUR NAME
ATTACH,REVSLIB,TRWREVS7600SYSTEM,ID=PTLREVS.
LIBRARY,REVSLIB.
REVSPRE.
EXIT,U.
REVSXQT.
                 7/8/9 CARD
     REVS INPUT CARDS
                 6/7/8/9 CARD
```

9.2.7.2 Building an Initial Data Base

The deck setup below illustrates a job which constructs and saves an ASSM on a catalogued file. To save the same ASSM on tape, the REQUEST card would be replaced by the appropriate tape REQUEST or STAGE card and the CATALOG card removed.

```
JOBNO,STMFZ,T77.                                         YOUR NAME
ATTACH,REVSLIB,TRWREVS7600SYSTEM,ID=PTLREVS.
LIBRARY,REVSLIB.
REQUEST,TAPE2,*PF.
REVSPRE.
EXIT,U.
REVSXQT.
CATALOG,TAPE2,YOURNAMEFORDATABASE,ID=YOURID.
                7/8/9 CARD
    REVS INPUT CARDS
                6/7/8/9 CARD
```

9.2.7.3 Updating a Data Base

The two deck setups below illustrate jobs which update an existing data residing on a catalogued file. The first deck setup does not save the resultant data base; the second deck setup does save the updated data base. The REQUEST, GETPF, and CATALOG cards should be replaced by the appropriate REQUEST or STAGE cards as described in [4] if the data base is maintained on tape rather than on a disk file.

```
JOBNO,STMFZ,T77.                                         YOUR NAME
ATTACH,REVSLIB,TRWREVS7600SYSTEM,ID=PTLREVS.
LIBRARY,REVSLIB.
REVSPRE.
RETURN,TAPE2.
GETPF,TAPE2,YOURNAMEFORDATABASE,ID=YOURID.
EXIT,U.
REVSXQT.
                7/8/9 CARD
    REVS INPUT CARDS
                6/7/8/9 CARD
                ----------------------------
JOBNO,STMFZ,T77.                                         YOUR NAME
ATTACH,REVSLIB,TRWREVS7600SYSTEM,ID=PTLREVS.
LIBRARY,REVSLIB.
REVSPRE.
RETURN,TAPE2.
REQUEST,TAPE2,*PF.
GETPF,TAPE2,YOURNAMEFORDATABASE,ID=YOURID.
EXIT,U.
REVSXQT.
CATALOG,TAPE2,YOURNAMEFORDATABASE,ID=YOURID.
                7/8/9 CARD
    REVS INPUT CARDS
                6/7/8/9 CARD
```

9.2.7.4 Executing and Saving a Simulator/Post Processor

The deck setup below constructs, executes, and saves a simulator and post processor. The files catalogued on the disk file will include the recording data base generated by the execution of the simulator since the SIMRUN precedes the SIMSAVE. To save the simulator files on tape rather than on a catalogued file, the REQUEST card should be replaced by an appropriate tape REQUEST or STAGE card as described in [4], and the CATALOG card removed.

```
JOBNO,STMFZ,T77.                                                YOUR NAME
ATTACH,REVSLIB,TRWREVS7600SYSTEM,ID=PTLREVS.
LIBRARY,REVSLIB.
REVSPRE.
RETURN,TAPE2.
GETPF,TAPE2,YOURNAMEFORDATABASE,ID=YOURID.
EXIT,U.
REVSXOT.
EXIT,U.
SIMRUN.
REQUEST,SIMFILE,*PF.
SIMSAVE.
CATALOG,SIMFILE,YOURNAMEFORSIMULATORFILE,ID=YOURID.
TESTRUN.
                 7/8/9 CARD
     REVS INPUT CARDS
                 6/7/8/9 CARD
```

9.2.7.5 Loading and Executing a Simulator

The deck setup below illustrates a job which loads a previously saved simulator and executes it. If the simulator had been saved on tape rather than on a catalogued file, then the ATTACH card would be replaced by the appropriate REQUEST or STAGE card as described in [4].

```
JOBNO,STMFZ,T77.                                                YOUR NAME
ATTACH,REVSLIB,TRWREVS7600SYSTEM,ID=PTLREVS.
LIBRARY,REVSLIB.
REVSPRE.
ATTACH,SIMFILE,YOURNAMEFORSIMULATORFILE,ID=YOURID.
SIMLOAD.
EXIT,U.
REVSXOT.
EXIT,U.
SIMRUN.
                 7/8/9 CARD
     REVS INPUT CARDS
                 6/7/8/9 CARD
```

10.0 INSTALLATION DEPENDENCIES

This section describes those portions of REVS, exclusive of the job control streams described in Section 9, which are installation dependent. The dependencies for each installation of REVS are discussed in separate subsections.

10.1 TI-ASC DEPENDENCIES

This section describes the REVS dependencies as they apply to the installations of REVS on the TI-ASC, both at the Naval Research Laboratory (NRL) in Washington, D.C., and at the Ballistic Missile Defense Advanced Technology Center (BMDATC) Advanced Research Center (ARC) in Huntsville, Alabama.

10.1.1 Character Set

The ASC installation of REVS uses the standard EBCDIC character set. All characters will print as shown in this document.

10.1.2 CALCOMP Plotting Symbols

The symbols plotted by the ASC installations of REVS agree with those shown in Figure 5-2.

10.1.3 Operating Modes

The on-line operating mode is available in the ARC ASC installation but not in the NRL ASC installation. There is no restriction on the number of times a user may use the on-line mode in the ARC ASC installation.

10.1.4 TI-PDL 2 Compiler

Simulators and post processors generated by ASC installations of REVS are compiled using the TI-PDL 2 compiler described in Volume 1 of [1]. No restrictions exist beyond those defined in that volume.

10.1.5 Linkage Editor

The standard ASC linkage editor [5] is used to link-edit the simulators and post processors generated by ASC installations of REVS. To insure proper link-editing, elements of the following element types, when used in simulating requirements, must be unique in the first eight characters: ALPHA, PERFORMANCE_REQUIREMENT, R_NET, SUBNET, SUBSYSTEM, VALIDATION_POINT.

10.2 CDC 7600 DEPENDENCIES

This section describes the REVS dependencies as they apply to the REVS installation on the CDC 7600 at the Ballistic Missile Defense Advanced Technology Center (BMDATC) Advanced Research Center (ARC) in Huntsville, Alabama.

10.2.1 Character Set

The ARC CDC 7600 installation of REVS uses the standard BCD character set. All characters will print as shown in this document with the following exceptions:

Character	CDC Graphic	ASCII Graphic	Hollerith Punch (026)	ASCII Punch (029)
double quote	≠	"	8-4	8-7
underscore	→	_	0-8-5	0-8-5
vertical bar	]	]	0-8-2	11-8-2
up-arrow	↑	'	11-8-5	8-5

10.2.2 CALCOMP Plotting Symbols

The symbols plotted by the ARC CDC 7600 installation of REVS agree with those shown in Figure 5-2.

10.2.3 Operating Modes

The ARC CDC 7600 installation of REVS allows the use of the on-line operating mode with the restriction that the user may use the on-line mode only once in any REVSXQT step.

10.2.4 TRW PASCAL Compiler

Simulators and post processors generated by the ARC CDC 7600 installation of REVS are compiled using a TRW PASCAL compiler which implements the PASCAL language as defined in [2]. Limitations of this compiler require that elements of the following element types, when used in simulating requirements, must be unique in the number of characters specified:

unique in 7 characters	unique in 10 characters
FILE	DATA (TYPE not ENUMERATION)
ENTITY_TYPE	ENTITY_CLASS
DATA (with type ENUMERATION)	MESSAGE
	INPUT_INTERFACE
	OUTPUT_INTERFACE
	values in a RANGE attribute

Also, the Boolean operators EQU and XOR defined in the syntax for RSL are not recognized by this PASCAL compiler and should not be used in the ARC CDC 7600 installation.

10.2.5 SCOPE Loader

The standard CDC SCOPE loader [6] is used to link-edit the simulators and post processors generated by the ARC CDC 7600 installation of REVS. To insure proper link-editing, elements of the following element types, when used in simulating requirements, must be unique in the first seven characters: ALPHA, PERFORMANCE_REQUIREMENT, R_NET, SUBNET, SUBSYSTEM, VALIDATION_POINT.

APPENDIX A

EXTENDED BNF NOTATION

Throughout this document, the syntax of the Requirements Statement Language (RSL) and the REVS Control Language (RCL) has been defined in a notation which is called extended BNF (Backus-Naur Form). This notation is a loosely formal grammatical representation of the syntax.

A grammar is composed (at least in part) of rules, or productions. Each production specifies a textual replacement; by starting with the chosen initial symbol, and substituting as necessary using the productions, all legitimate forms (commands) of the language may be developed.

Three general classes of symbols appear in productions, namely terminal symbols, non-terminal symbols, and meta-linguistic symbols.

- Terminal symbols are those characters and character strings which will actually appear in a language statement. Examples of terminal symbols for RSL include words such as "STRUCTURE", "TERMINATE", any of the element names, and characters such as "." and "(". Any symbol which is not a non-terminal or meta-linguistic symbol is by default a terminal symbol. The terminal symbols may be grouped into two classes.

  - Individual symbols, such as punctuation marks (e.g., "(") and keywords (e.g., "TERMINATE"), are those symbols which represent themselves. That is, they appear in the BNF exactly as they will appear in RSL or RCL. Individual symbols which are words will be written in all capitals.

  - Class Symbols, such as "name" and "comment", are those symbols which denote a whole group of terminals with a formation rule to define the constitution of the class. Any member of the class may appear in the language statement or command. Class symbols will be written in lower case letters. Optional prefixes giving semantic information may be used with class symbols. An example of this is "element-name" which means that any "name" designating an "element" may be used.

- Non-terminal symbols name other productions in the language and will always be written enclosed in corner brackets. For example <set definition> is a non-terminal symbol in the RADX RCL syntax. Examples of non-terminals from RSL include <new element definition> and <relation declaration>.

APPENDICES

- Meta-linguistic symbols are those characters used to write productions. The character "<" and ">" are such characters, and are used to denote non-terminal symbols. The meta-linguistic symbols in addition to corner brackets ("<" and ">") used in this document are interpreted as follows:

  - Braces ("{" and "}") are used to indicate possible repetition of the phrase written within the braces. The subscript following the closing brace gives the minimum number of repetitions allowed; likewise, the superscript gives the maximum. Thus

    $$\left\{\text{<node>}\right\}_{1}^{n}$$

    means that 1 or more (with the upper bound indefinite) occurrences of <node> may be used at that point.

  - Brackets ("[" and "]") are a shorthand for $\left\{ \dots \right\}_{0}^{1}$, indicating that the enclosed phrase may be used once or omitted. For example,

    [MODIFY] element-type-name element-name.

    means that the terminal MODIFY may be omitted.

  - More than one phrase within brackets or braces (on different lines), denote that exactly one of the phrases is to be chosen on each repetition. The choice is completely independent from one repetition to the next; thus,

    $$a \left\{\begin{matrix} b \\ c \end{matrix}\right\}_{1}^{2} d$$

    generates the following phrases:

    abd
    acd
    abbd
    abcd
    acbd
    accd

  - The symbol "::=" is used to denote the definition of a non-terminal. All productions are of the form

    <non-terminal>::= phrase

    where <non-terminal> is thereby defined to be "phrase".

  - The symbol "|" is used to designate alternatives. If <a or b> is to be defined to be A or B, the production to express this would be

    <a or b>::= A|B

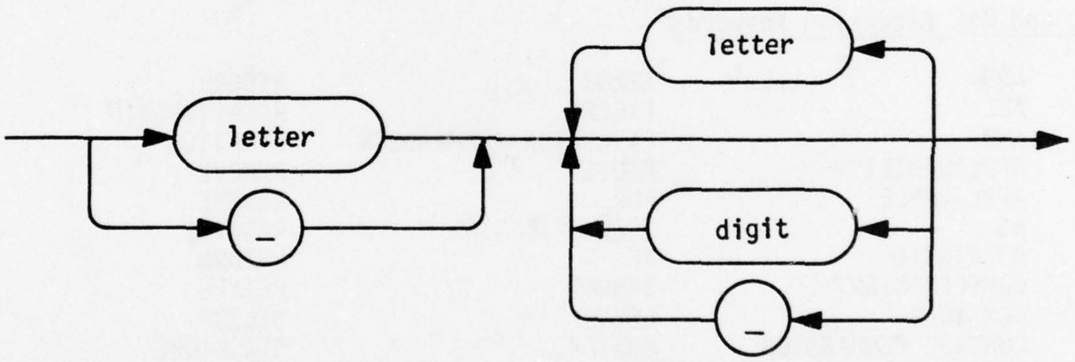
APPENDIX B

GENERAL SYNTAX RULES

The RSL and RCL grammars have, as their terminal symbols, words, punctuation marks, numbers, text strings, and comments. Each of these terminals is discussed in the following paragraphs.

Words

The word is a class symbol which corresponds to the intuitive definition of a word in English. It is a string of characters starting with a letter or underscore (except with RNETGEN which allows only a letter as the first character), and continuing with letters, digits or underscores (_). This definition is shown below in the form of a syntax diagram.



The word is terminated by one or more blanks or punctuation marks (see below). Note that the end of a card or line is defined to be an extra character which is read as a blank; therefore, no word may be split over a card boundary. No maximum length for a word is specified in the language definition (BNF or syntax diagram forms), however words are restricted to a maximum length of 60 characters.

Words are divided into three subclasses: reserved words, optional words, and names.

Reserved Words - Words which appear as keywords in the BNF and syntax diagram definitions of RSL and RCL are reserved for the use designated in the definitions. The REVS user may not redefine these words for his own use. In addition, the user should not use PASCAL keywords except in BETAs, GAMMAs, and TESTs.

Listed below are the RSL, RCL and PASCAL reserved words.

Executive RCL Keywords

(Since the Executive scans REVS input for Executive Control Statements, caution should be used in forming Function Control Statements so as not to use these keywords to inadvertently form Executive RCL at the beginning of a line image.)

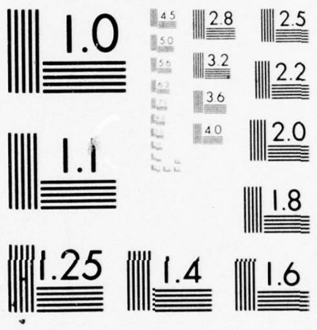
ADDFILE	LOG	RSLXTND
ALL	NEWPAGE	SIMDA
BOTH	OFFLINE	SIMGEN
EXECRCL	ONLINE	SIMXQT
FEND	ONLY	STEP
FUNCTION	OUTPUT	STOP
GO	RADX	TESTER
IMPLIED	RNETGEN	TRANSPARENT
JOB	RSL	

RSL and RSL Extension Keywords

ADD	ERROR	RECORD
ALL	EXCEPT	RELATIONSHIP
AND	EXTENSION_PERMISSION	RELATION
APPLICABILITY	FALSE	REMOVE
APPLICABLE	FOR	RENAME
AS	IDENTIFICATION	RESCIND
ATTRIBUTE	IF	RETURN
COMPLEMENTARY	INSERT	RETYPE
CONSIDER	LEVEL	SELECT
CONTROL_PERMISSION	MODIFY	STRUCTURE
DEFINE	MOD	SUBJECT
DELETE	NET	SUCH
DIV	NOT	TERMINATE
DO	OBJECT	THAT
EACH	OR	TRUE
ELEMENT_TYPE	OTHERWISE	VALUE
END	PATH	XOR
EQU	PERMISSION	

AD-A046 572 TRW DEFENSE AND SPACE SYSTEMS GROUP HUNTSVILLE ALA F/G 9/2
REVS USERS MANUAL. SREP FINAL REPORT. VOLUME II.(U)
AUG 77 M E DYER, L J GUNTHER, R W SMITH DASG60-75-C-0022
UNCLASSIFIED TRW-27332-6921-026-VOL-2 NL

4 OF 5
AD
A046572



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RADX RCL Keywords

ALL	HIERARCHY	REFERS
ANALYZE	IMPLIED	RELATION
AND	IN	RELATIONSHIP
ANY	IS	RSL
APPEND	LIST	SEQUENCE
ATTRIBUTE	MAP	SET
BETA	MINUS	STRUCTURE
BY	MULTUPLE	SUCH
COMPLEMENTARY	NO	SUMMARY
DATA_FLOW	NONE	THAT
ELEMENT_TYPE	NOT	USING
FROM	OR	WHERE
GAMMA	PLOT	WHICH
GROUP	PRIMARY	WIDTH
HEIGHT	PUNCH	WITH
HIER	REFERRED	WITHOUT

SIMGEN Keywords

RCL

ALL	IDENT	R_NETS
BETA	IDENTIFICATION	SIMULATION
EXCLUDE	INCLUDE	SIMULATOR
GAMMA	IS	TYPE
ID	R_NET	

BETA/GAMMA FILE Access

CREATE	FIRST	RECORD
DESTROY	FOR	SELECT
DO	FROM	SUCH
EACH	NEXT	THAT
ENDFOREACH		

TEST Recording Access

DO	FOR	RECORDING
EACH	FROM	RETRIEVE
ENDFOREACH	NEXT	SUCH
FIRST	RECORD	THAT

PASCAL

AND	FUNCTION	PROGRAM
ARRAY	GO TO	RECORD
BEGIN	IF	REPEAT
CASE	IN	SET
CONST	LABEL	THEN
DIV	MOD	TO
DO	NIL	TYPE
DOWN TO	NOT	UNTIL
ELSE	OF	VAR
END	OR	WHILE
FILE	PACKED	WITH
FOR	PROCEDURE	

PDL 2 Extensions to PASCAL

ACCESS	COMMON	MACRO
ALIGNED	ESCAPE	XOR
BY	EQU	

SIMXQT RCL Keywords

END	IS	SIMULATOR
ID	RUN	START
IDENT	SIMULATION	TIME
IDENTIFICATION		

SIMDA RCL Keywords

ALL	PERFORMANCE_REQUIREMENT	TEST
EXCEPT	PERFORMANCE_REQUIREMENTS	

Optional Words - Optional words are defined in RSL as part of a relation definition (see Section 8.0, Extending the Language (RSLXTND Function)). Once defined, an optional word may be used anywhere in the RSL input stream and is essentially ignored by the RSL translation function. (Note: Optional words appearing in conditionals, comments, and text strings will be stored in the ASSM. Optional words should not be used in conditions or text strings containing PASCAL code; they will result in simulator post/processor compilation errors.) Optional words can appear only in input to the translator; they cannot appear in RCL.

If a relation definition is deleted from the language, the optional word is deleted. However, an optional word may be associated with more than one relation and, thus, as long as one relation associating the optional word remains defined, the optional word may be used. Examples of possible optional words are: BY, FROM, IN, OF, TO, WITH. All optional words currently defined appear in the RSL concept definitions appearing in Appendix D, Section 3.

Names - All other words appearing in the definition of RSL and RCL are interpreted as names which take on user specifiable semantic interpretation. Throughout the BNF and syntax diagrams appearing in the appendices, prefixes are added to a terminal symbol name to indicate the semantic interpretation which is implied by using a name in the indicated position.

Punctuation Marks

Punctuation marks are individual symbols which correspond roughly to English punctuations. There are two classes of punctuation marks recognized by REVS.

The following punctuation marks appear explicitly in the RSL and RCL syntax and are therefore reserved for those uses alone:

. ( ) < > <= >= = <> + - * /

The other class of punctuation marks is ignored by REVS and may be used to improve readability:

; , :

These punctuation marks cannot be used in a conditional.

Numbers

Number is a class symbol in RSL and RCL which has the formation rules for numbers that are found in PASCAL. Specifically, the following rules apply:

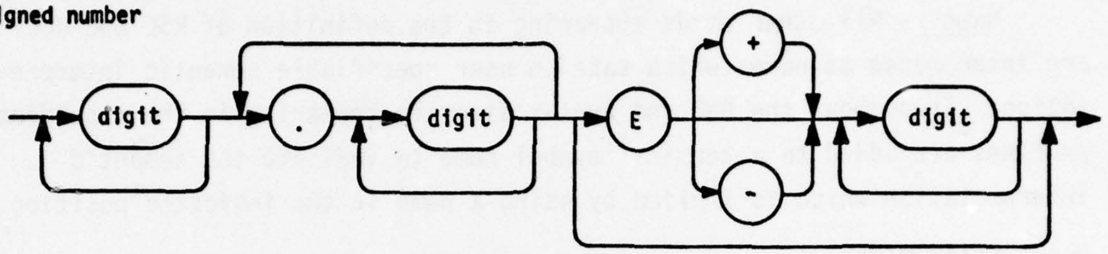
```
<signed number>::= <sign> <unsigned number>

<sign>::= + | -

<unsigned number>::= <unsigned integer> | <unsigned real>

<unsigned integer>::= <digit sequence>
```

$$\text{<digit sequence>::=} \left\{\text{digit}\right\}_1^n$$

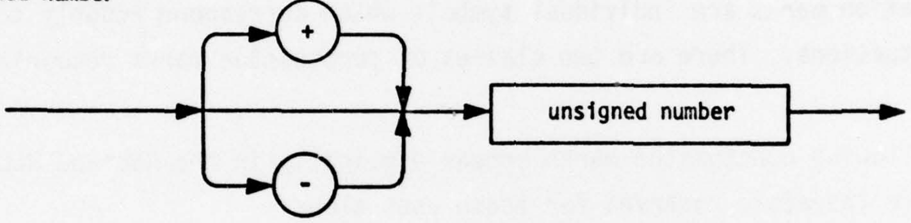
```
<unsigned real>::= <unsigned integer> . <digit sequence>
    | <unsigned integer> . <digit sequence> E <scale factor>
    | <unsigned integer> E <scale factor>
<scale factor>::= <unsigned integer> | <sign> <unsigned integer>
```

These syntax rules are summarized in the following diagrams.

unsigned number



signed number



Note that if the number contains a decimal point, at least one digit must precede and succeed the point. Also, no comma may occur in a number.

Text Strings

The text string is a class symbol. It consists of any sequence of characters surrounded by double quotes (i.e., " ... ").

Comment

Comment is a class symbol which consists of any sequence of characters beginning with (* and ending with *) (i.e., (* ...*)). Comments in RSL may be placed only where specified in the syntax. Comments may be entered through RNETGEN only where legal in the corresponding RSL syntax. Comments may appear anywhere in a RADX command; and comments are not legal in any other portion of RCL.

APPENDIX C

REVS EXECUTIVE SUMMARY

C.1 EXECUTIVE RCL SYNTAX

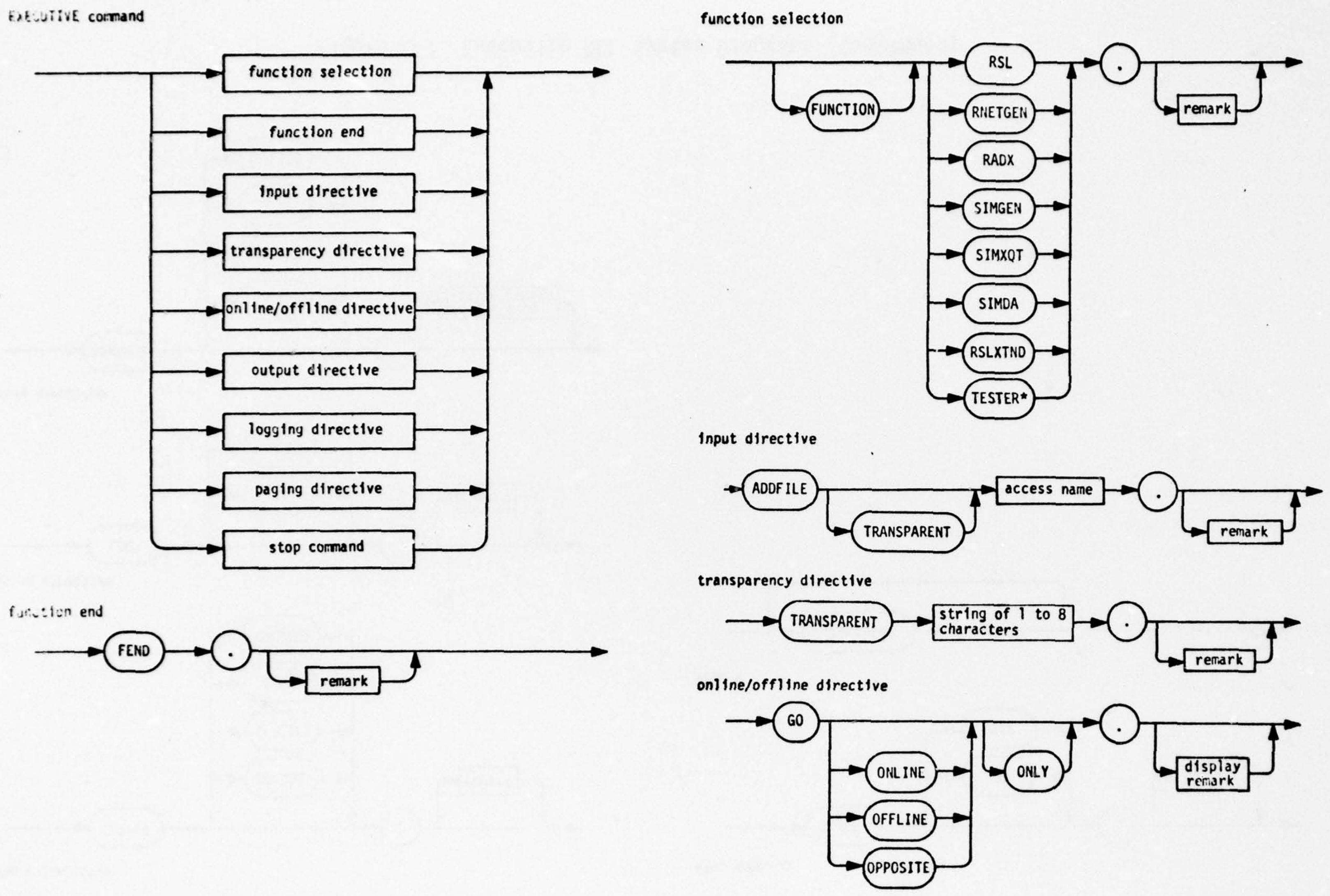
Table C.1 contains a description of the syntax of Executive RCL in the Backus-Naur Form described in Appendix A. An underline is used in this syntax description to identify the assumed keyword when an optional keyword is omitted. For each syntax production or set of productions for the Executive RCL commands, this table also identifies the number of the section in this document where the command is described.

Figure C-1 shows the syntax of Executive RCL in diagrammatic form.

Table C.1 Executive RCL Index

EXECUTIVE RCL SYNTAX	SECTION NUMBER
<REVS Executive command>::= **<function selection> | <function end> | <transparency declaration>** **| <input directive> | <online/offline directive> | <logging directive>** **| <output directive> | <paging directive> | <stop command>**	4.0
<function selection>::= [FUNCTION] <function name>. [remark] <function name>::= RSL | RNETGEN | RADX | SIMGEN | SIMXQT | SIMDA | RSLXTND | TESTER*	4.1
<function end>::= FEND. [remark]	4.1
<transparency declaration>::= TRANSPARENT string-of-1-to-8 characters. [remark]	4.2.1
<input directive>::= ADDFILE [TRANSPARENT] access-name. [remark]	4.2.2
<online/offline directive>::= GO [ONLINE OFFLINE OPPOSITE] [ONLY]. [display-remark]	4.3
<logging directive>::= LOG [ALL EXECRCL]. [remark]	4.4.1
<output directive>::= OUTPUT [ONLINE OFFLINE BOTH IMPLIED]. [remark]	4.4.2
<paging directive>::= NEWPAGE [OFFLINE ONLINE BOTH IMPLIED]. [offline-page-titling-remark]	4.4.3
<stop command>::= STOP [JOB STEP]. [display-remark]	4.5

*The TESTER function is reserved for software development use only.



*the TESTER function is reserved for software development use only.

Figure C-1 Executive RCL Syntax Diagrams

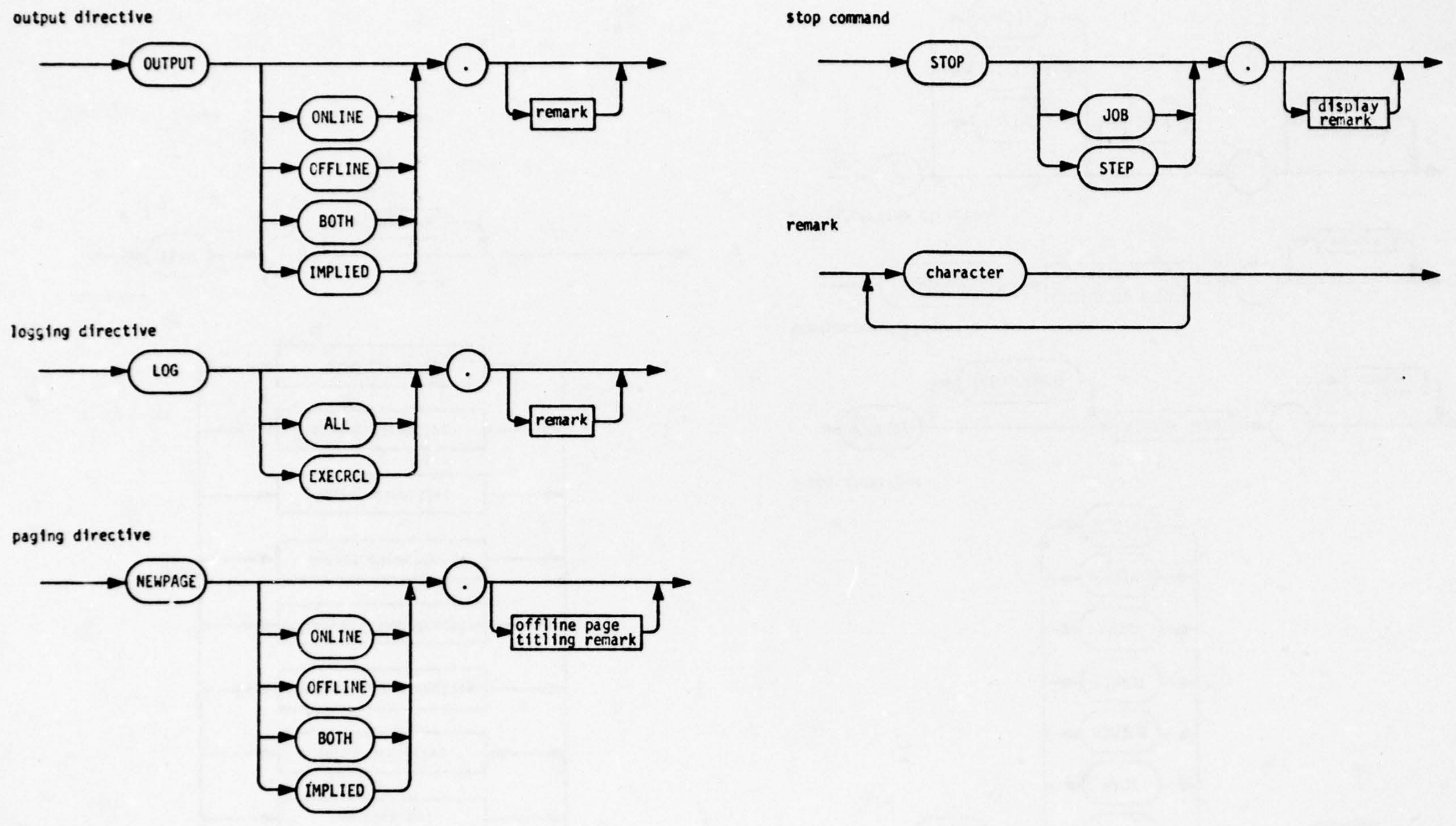


Figure C-1 Executive RCL Syntax Diagrams (Continued)

C.2 EXECUTIVE MESSAGES

The messages output by the REVS Executive are identified and described below. These messages appear both on REVS.LOG and REVS.OUT except as noted.

XX 000 REVS BASELINE VERSION = X,(DATE = __/__/__, TIME = __:__:__).

Identifies the version of REVS being executed, showing the baseline number, and the data and time the baseline was created. It appears at the beginning of the print file. Executive output follows this line.

XX 001 FUNCTION function-name INITIATED.

Identifies the REVS function being initiated, and indicates a change of state from executive to function. On REVS.OUT, this line is padded with asterisks to highlight the state change. Function output follows this line.

XX 002 FUNCTION function-name COMPLETED.

Identifies the REVS function which is terminating and indicates a change of state from function to executive. On REVS.OUT, this line is padded with asterisks to highlight the state change. Executive output follows this line.

XX 003 DATA BASE OPEN FAILURE.

Indicates that the ASSM cannot be opened for use and therefore causes an immediate REVS termination.

XX 004* FEND VIOLATION, PROGRAM ERROR: ABORT.

Indicates a function attempt to read past the FEND statement. This should only occur as a result of a system/hardware error.

XX 005* STOP VIOLATION, PROGRAM ERROR: ABORT.

Indicates an executive attempt to read past the STOP statement. This should only occur as a result of a system/hardware error.

XX 006 NON REVS-EXEC-RCL STATEMENT: IGNORED.

Indicates that non-executive RCL statements (i.e., Function Control Statements) have been encountered while in the executive state. This message appears only once per executive state regardless of the amount of irrelevant input. This may be a result of the misplacement or omission of Executive RCL statements or can occur if a function prematurely terminates before reading all its input.

XX 007 REVS COMPLETED: NORMAL TERMINATION.

Indicates normal termination of REVS. All output is complete.

*This message appears only on REVS.LOG.

XX 008 REVS COMPLETED: ABNORMAL TERMINATION.

Indicates abnormal termination of REVS. Other messages indicating the source of the error may be found in the ASC Job Activity File (SYS.JATF) at the front of the run, or in the REVS.OUT listing. All output should be completed.

XX 009 PDS 2 RUN-TIME ERROR: ABORT IN PROCESS $\left\{\begin{array}{l}\text{HARDWARE/STACK OVERFLOW}\\ \text{UTILITY/LIBRARY}\\ \text{BOUNDS CHECK}\end{array}\right\}_{1}^{1}$.

Indicates premature termination. The suffix of the message gives a general classification of the error detected:

- HARDWARE/STACK OVERFLOW means a hardware interrupt or PDS stack overflow has occurred. The hardware interrupt means hardware error, program error, or time out and is clarified on REVS.OUT. Time can be increased on the REVSXQT macro.
- UTILITY LIBRARY means an error was detected by the PDS utility library, and is clarified on REVS.OUT.
- BOUNDS CHECK means a program error was detected, is clarified on REVS.OUT and probably indicates a hardware error.

XX 010* XXHALT TERMINATION REQUESTED.

Indicates a programmed emergency termination and can only occur by hardware error.

XX 011* ADDFILE COMPLETED: RECORDS READ = number.

Indicates completion of the reading of an alternate input file. The number of records read is printed.

XX 012 REVS CHANGING MODE TO ONLINE.

Indicates that the Executive is changing the operating mode to on-line. Subsequent output is to the on-line console unless specifically routed off-line with an OUTPUT statement.

XX 013 REVS CHANGING MODE TO OFFLINE.

Indicates that the Executive is changing the operating mode to off-line. Output routing is governed by the last OUTPUT statement.

XX 014 PREVIOUS GO ONLY ACTIVATED STOP ON THIS GO STATEMENT.

Indicates a GO statement was encountered following one with the ONLY option and that it is being interpreted as a STOP statement.

XX 015 REVSGRAPH RELEASE OMISSION, PROGRAM ERROR: EXEC CORRECTED.

Indicates synchronization error between the Executive and a REVS function over control of the on-line console. This condition can only occur as the result of a hardware error.

---

*This message appears only on REVS.LOG.

XX 016 REVS TERMINATING: GO OFFLINE, STOP IMPLIED.

Indicates REVS is terminating while still in the on-line mode. Previous messages on REVS.OUT explain why. This message appears on the on-line console only.

XX 017* ONLINE USER HAS REQUESTED INTERRUPT.

Indicates the on-line user has requested a function interrupt with the trackball at a page wrap opportunity. Reaction to the request is function dependent.

XX 018 DEFAULT ONLINE IDENTIFICATION USED: user-name.

Indicates that the user has failed to provide a personal identification notice in the comment field of the GO ONLINE statement to be shown on the introductory REVS display on the ANAGRAPH on-line console. A default identification is thus displayed which is the user name from the job card.

XX 019* ONLINE CONSOLE ACQUIRED SUCCESSFULLY.

Indicates the actual time that REVS achieved the on-line mode. Comparison with the time of the previous message (XX 012) reveals that the delay is due to non-availability of any graphics communication or to console contention.

XX 020* user-identification NOW ACTIVE ONLINE.

Identifies the user and time of initial on-line response. Comparison with the time of the previous message (XX 019) reveals the extent of a REVS idle condition waiting for user activity.

XX 021* NUMBER OF CARDS PUNCHED BY REVS IS number.

Indicates the number of cards punched by REVS as a result of RADX punch statements. This message occurs only if punched output is produced (number of cards > 0).

XX 022* NUMBER OF STRUCTURES PLOTTED = number, MAXX = number, MAXY = number.

Indicates the number of structures plotted by RNETGEN and/or RADX and the maximum X and Y plot size in inches. This is formatted to ease the task of manually completing plot requests at the ARC. Plotting is automatically completed at NRL.

*This message appears only on REVS.LOG.

APPENDIX D

RSL SUMMARY

D.1 RSL SYNTAX

Table D.1 contains a description of the syntax of RSL in the BNF notation described in Appendix A. This table contains the syntax for only that part of RSL used for developing requirements; the syntax for extending the language is documented in Appendix J. For each syntax production or set of productions for the RSL commands, this table also identifies the number of the section in this document where the command is described.

Figure D-1 shows the syntax of RSL in diagrammatic form. Since the formal syntax for RSL allows statements which will result in semantic errors, some of the semantic rules have been incorporated in the statement of the grammar in order to aid the user of RSL.

Table D.1 RSL Index

RSL SYNTAX	SECTION NUMBER
<RSL command>::=	
<new element definition>	
| <element modification>	
| <element deletion>	
| <element rename>	
| <element retype>	5.1
<new element definition>::=	
[DEFINE] element-type-name element-name [comment].	
$\{$[INSERT] <element definition sentence>$\}_0^n$	
<element definition sentence>::=	
<attribute declaration>	
| <relation declaration>	
| <path declaration>	
| <structure declaration>	5.1.1
<attribute declaration>::=	
attribute-name $\left\{\begin{array}{l}\text{value-name}\\ \text{number}\\ \text{text-string}\end{array}\right\}_1^1$ [comment].	5.1.1.2
<relation declaration>::=	
relation-name [relation-optional-word]$\{$[element-type-name] element-name [comment]$\}_1^n$.	5.1.1.3
<path declaration>::=	
PATH $\{$<element node>$\}_1^n$ END [comment].	5.1.1.5
<structure declaration>::=	
STRUCTURE $\{$<node>$\}_2^n$ END [comment].	
<node>::=	
<element node>	
| <terminator>	
| <and node>	
| <or node>	
| <consider-or node>	
| <for-each node>	
| <select node>	
<element node>::=	
[element-type-name] element-name [comment]	
<terminator>::=	
TERMINATE [comment]	
| RETURN [comment]	
<and node>::=	
DO [comment] <branch> AND$\{$<branch>$\}_1^n$ END	
<branch>::=	
$\{$<node>$\}_1^n$	
<or node>::=	
IF [comment] <conditional branch>	
$\{$OR <conditional branch>$\}_0^n$	
OTHERWISE [<branch>] END	5.1.1.4

Table D.1 RSL Index (Continued)

RSL SYNTAX	SECTION NUMBER
<conditional branch>::=	
[unsigned integer] <condition> <branch>	
<consider-or node>::=	
<consider-data>	
| <consider-entity-class>	
<consider-data>::=	
CONSIDER [DATA] enumerated-data-name	
IF [comment] <consider-data branch>	
$\{$OR <consider-data branch>$\}_1^n$	
END	
<consider-data branch>::=	
(enumeration-value-name $\{$OR enumeration-value-name$\}_0^n$) [<branch>]	
<consider-entity-class>::=	
CONSIDER [ENTITY_CLASS] entity-class-name	
IF [comment] <consider-entity branch>	
$\{$OR <consider-entity branch>$\}_1^n$	
END	
<consider-entity branch>::=	
(entity-type-name $\{$OR entity-type-name$\}_0^n$) [<branch>]	
<for-each node>::=	
FOR EACH $\{$ [FILE] file-name [RECORD]	
[ENTITY_TYPE] entity-type-name	
[ENTITY_CLASS] entity-class-name $\}_1^1$ [SUCH THAT <condition>]	
DO [comment] $\{$ [ALPHA] alpha-name [comment]	
[SUBNET] subnet-name [comment] $\}_1^1$ END	
<select node>::=	
SELECT $\{$ [ENTITY_CLASS] entity-class-name	
[ENTITY_TYPE] entity-type-name $\}_1^1$ SUCH THAT <condition> [comment]	
<condition>::=	
(<Boolean expression>)	
<Boolean expression>::=	
<simple Boolean> $\{$<B op> <simple Boolean>$\}_0^n$	
<simple Boolean>::=	
<Boolean term >$\{$OR <Boolean term>$\}_0^n$	
<Boolean term>::=	
<Boolean factor> $\{$AND <Boolean factor>$\}_0^n$	
<Boolean factor>::=	
<Boolean> [<rel op> <Boolean>]	
| <arithmetic expression> <rel op><arithmetic expression>	
<Boolean>::=	
[NOT] <Boolean primary>	
<Boolean primary>::=	
TRUE	
| FALSE	
| Boolean-data-name	
| (<Boolean expression>)	5.1.1.4

Table D.1 RSL Index (Continued)

RSL SYNTAX	SECTION NUMBER
<arithmetic expression>::= [<ad op>] <arithmetic term> $\{$<ad op> <arithmetic term>$\}_0^n$ <arithmetic term>::= <arithmetic factor> $\{$<mul op> <arithmetic factor>$\}_0^n$ <arithmetic factor>::= unsigned number | arithmetic-data-name | (<arithmetic expression>) * <B op>::= EQU | XOR <rel op>::= = | < | > | < = | > = | <> <ad op>::= + | - <mul op>::= * | / | DIV | MOD	5.1.1.4
<element modification>::= [MODIFY] element-type-name element-name [comment]. $\left\{\begin{array}{l}\text{[INSERT] <element definition sentence>}\\ \text{<attribute declaration removal>}\\ \text{<relationship declaration removal>}\\ \text{<structure declaration removal>}\\ \text{<path declaration removal>}\end{array}\right\}_0^n$	5.1.2
<attribute declaration removal>::= REMOVE attribute-name.	5.1.2.3
<relationship declaration removal>::= REMOVE relation-name [relation-optional-word] $\{$[element-type-name] element-name$\}_1^n$.	5.1.2.5
<structure declaration removal>::= REMOVE STRUCTURE.	5.1.2.7
<path declaration removal>::= REMOVE PATH.	5.1.2.9
<element deletion>::= DELETE element-type-name element-name.	5.1.3
<element rename>::= RENAME element-name AS new-element-name [comment].	5.1.4
<element retype>::= RETYPE element-name AS element-type-name.	5.1.5

*See Section 10 for installation dependent restrictions.

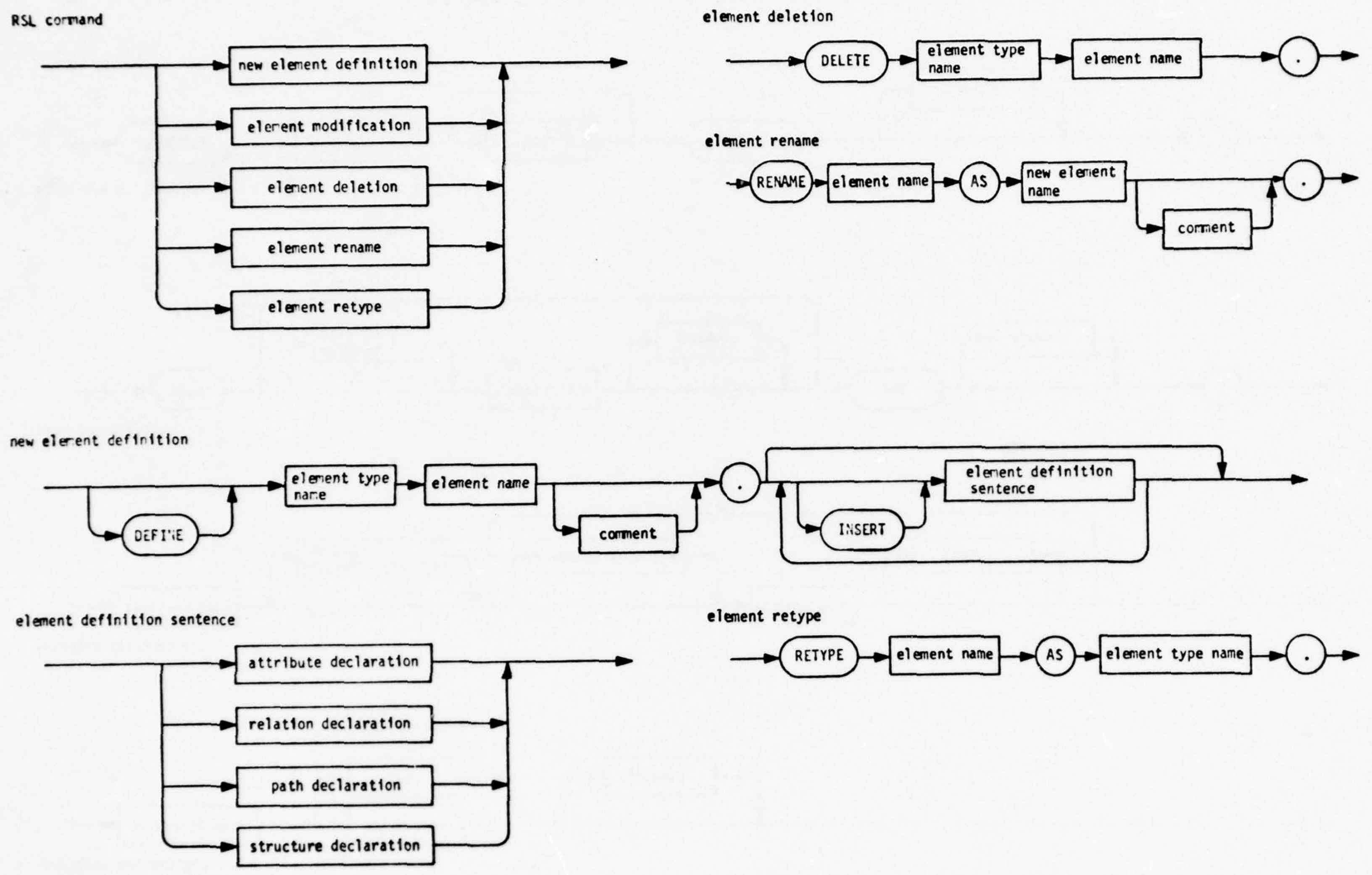


Figure D-1 RSL Syntax Diagrams

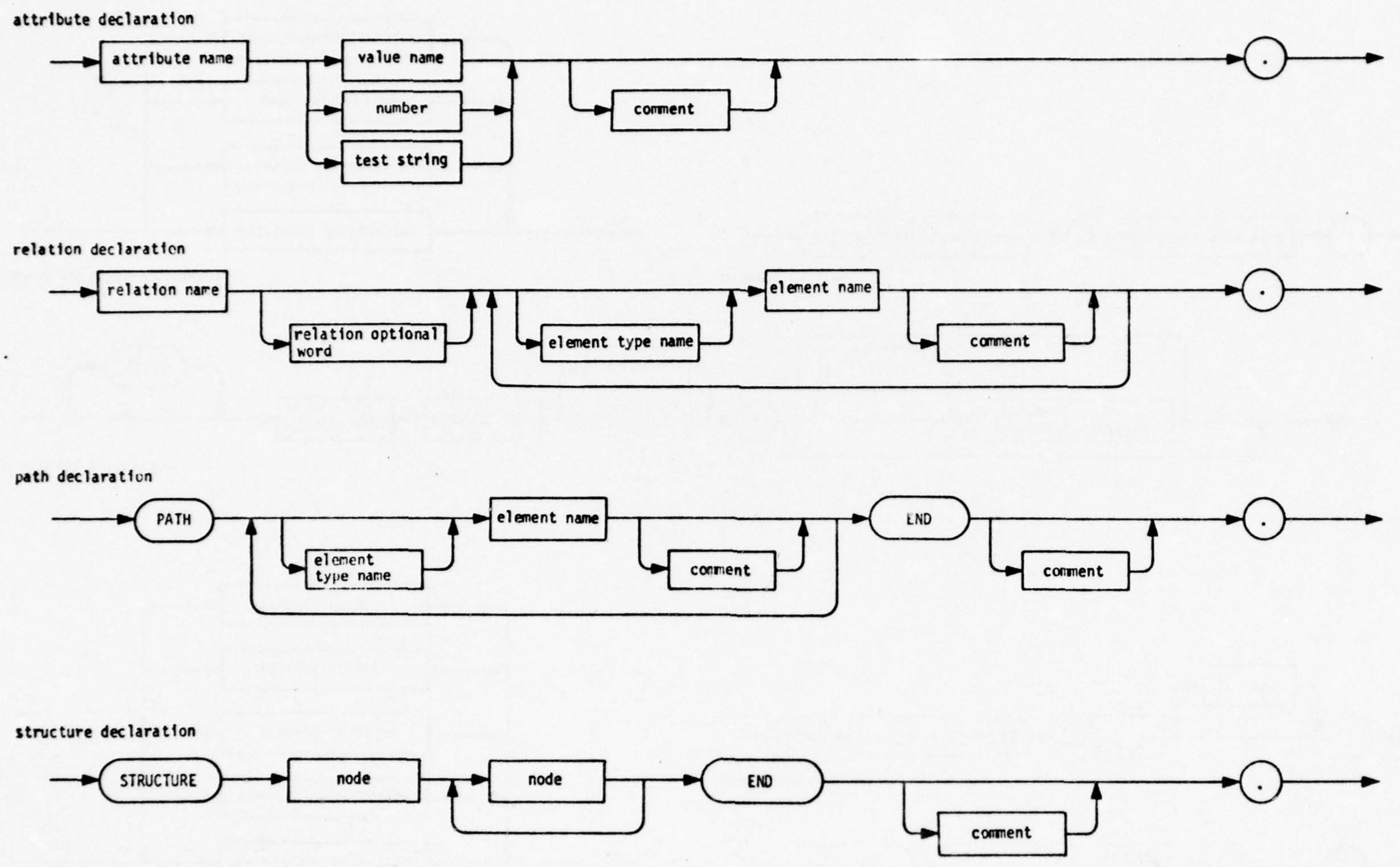


Figure D-1 RSL Syntax Diagrams (Continued)

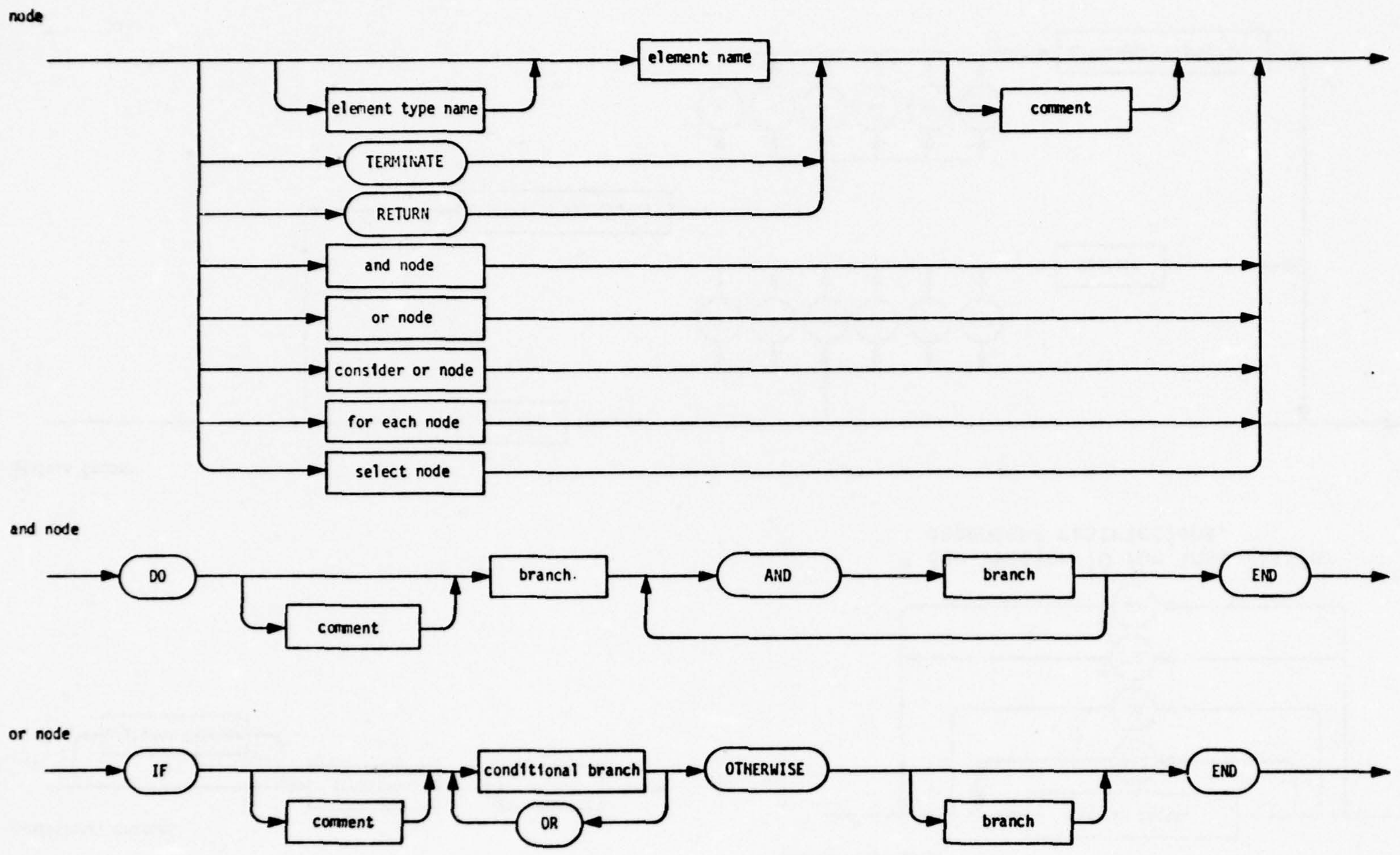
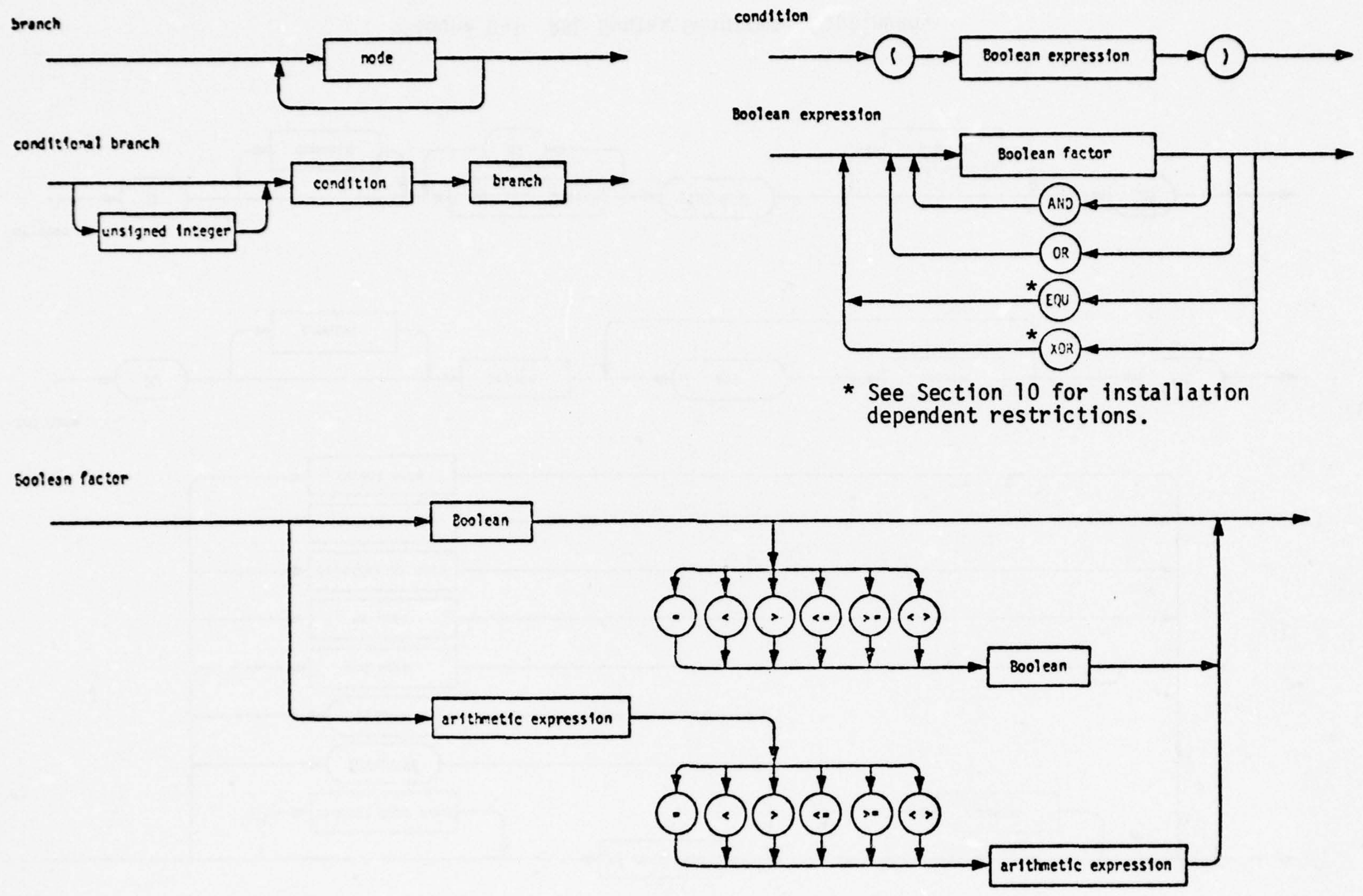


Figure D-1 RSL Syntax Diagrams (Continued)



* See Section 10 for installation dependent restrictions.

Figure D-1 RSL Syntax Diagrams (Continued)

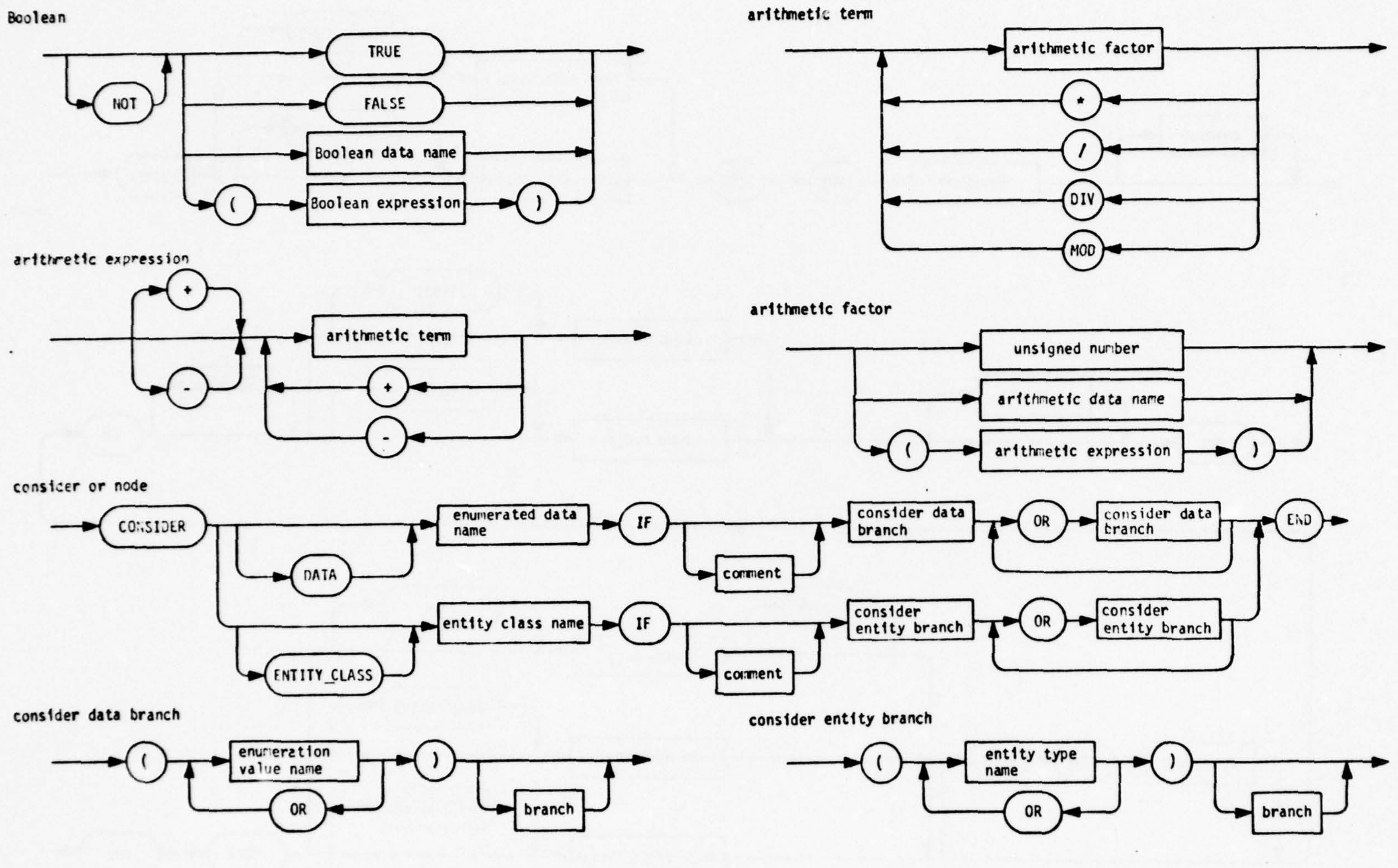


Figure D-1 RSL Syntax Diagrams

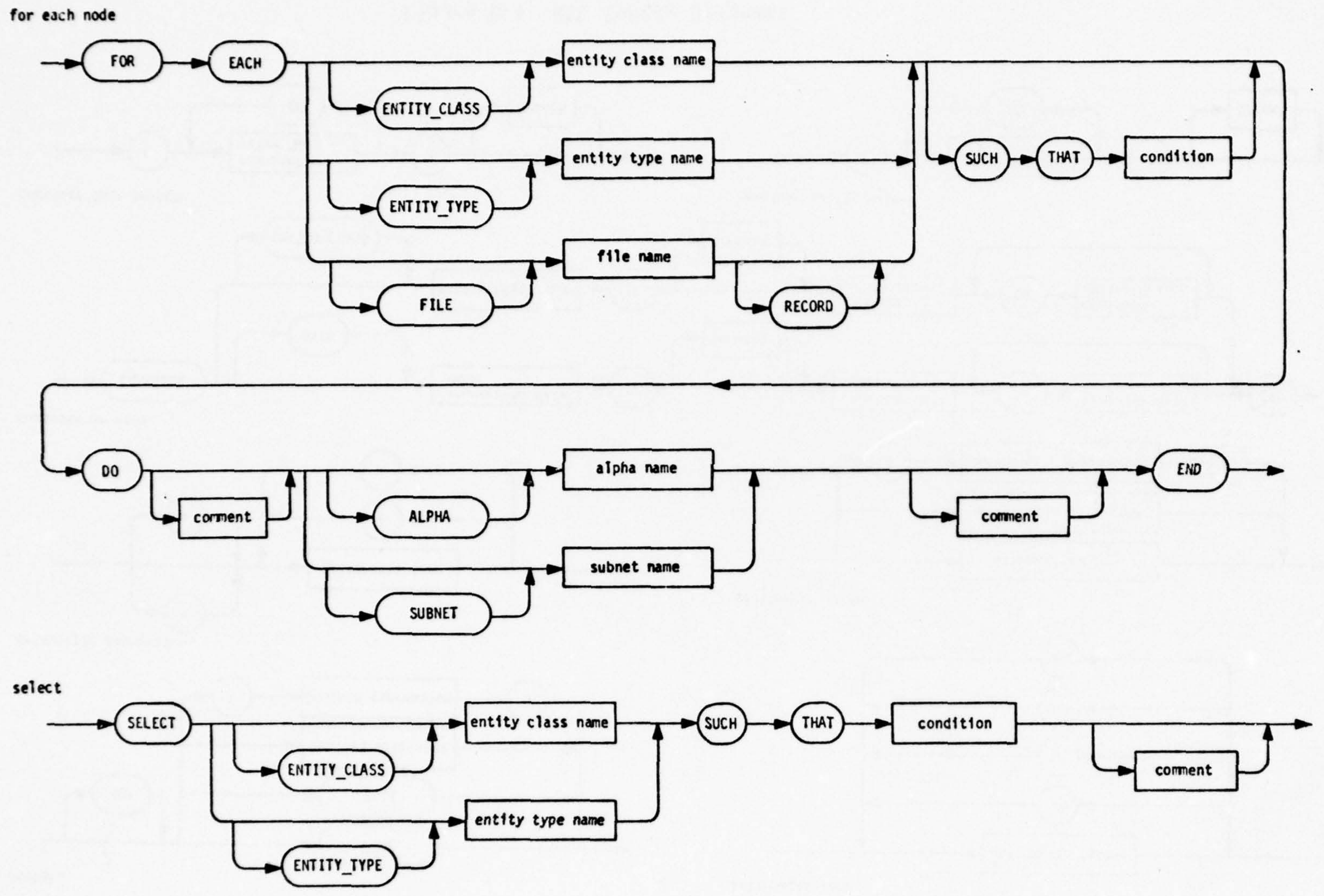


Figure D-1 RSL Syntax Diagrams (Continued)

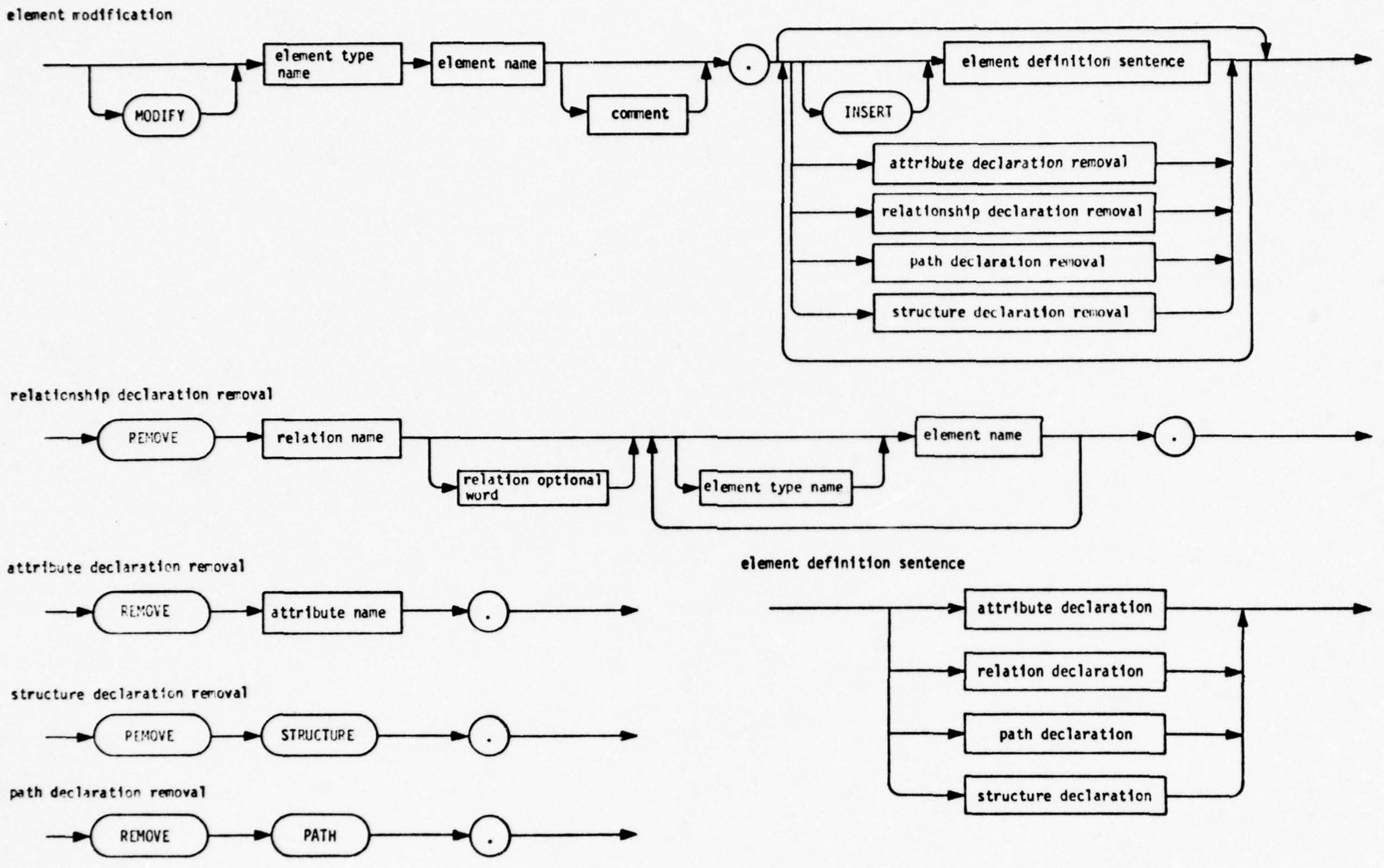


Figure D-1 RSL Syntax Diagrams (Continued)

D.2 RSL CONCEPT CROSS-REFERENCE

Table D.2 provides a cross-reference between the standard RSL element types, relationships, attributes, and structures. The element types are partitioned into segments corresponding to the segments which introduce and discuss them in Section 3 of this document. For relationships, attributes, and structures the defining segment is indicated by an X.

For each element type additional entries in Table D.2 indicate whether the element type may be the subject (S) or object (O) of each relationship and also whether the element type is an applicable element type for each attribute (*). If an element type may appear as an element node on an RSL structure, the appropriate table entry contains the entry A.

Table D.2 RSL Concept Cross-Reference Chart

ELEMENTS	RELATIONSHIPS																							ATTRIBUTES																					STRUC-TURE	
	ASSOCIATES	COMPOSES	CONNECTS	CONSTRAINS	CONTAINS	CREATES	DELAYS	DESTROYS	DOCUMENTS	ENABLES	EQUATES	FORMS	IMPLEMENTS	INCLUDES	INCORPORATES	INPUTS	MAKES	ORDERS	OUTPUTS	PASSES	RECORDS	SETS	TRACES	ALTERNATIVES	ARTIFICIALITY	BETA	CHOICE	COMPLETENESS	DESCRIPTION	ENTERED_BY	GAMMA	INITIAL_VALUE	LOCALITY	MAXIMUM_TIME	MAXIMUM_VALUE	MINIMUM_TIME	MINIMUM_VALUE	PROBLEM	RANGE	RESOLUTION	TEST	TYPE	UNITS	USE	PATH	STRUCTURE
MANAGEMENT:									X		X		X		X								X	X	X		X	X	X	X								X								
DECISION									0		0		S										S/0	*			*	*	*	*								*								
ORIGINATING_REQUIREMENT									0		0		S		S/0								S					*	*	*																
SOURCE									S		0																	*	*	*																
SYNONYM											S																																			
UNSTRUCTURED_REQUIREMENT									0		0		S										0					*	*	*																
VERSION									0		0		0										0					*	*	*																
ALPHA:																										X					X															
ALPHA						S		S	0		0	S	S			S			S			S	0		*	*		*	*	*	*															A
REQUIREMENTS NETWORK:							X			X																																				X
EVENT							0		0	S	0		S										0		*			*	*	*															A	A
R_NET									0	0	0		S										0		*			*	*	*																
SUBNET									0		0		S										0		*			*	*	*																A
DATA:	X	X	X		X	X		X				X		X		X	X	X	X	X	X	X										X	X		X		X		X	X		X	X	X		
DATA	0				0		S		0		0		S	S/0		0	S	S	0		0		0		*			*	*	*		*	*		*		*		*	*		*	*	*		
ENTITY_CLASS	S	0				0		0	0		0		S										0		*			*	*	*																
ENTITY_TYPE	S	S							0		0		S									0	0		*			*	*	*																
FILE	0				S				0		0		S			0	S	0	0		0		0		*			*	*	*			*													
INPUT_INTERFACE			S						0	S	0		S							S			0		*			*	*	*																A
MESSAGE									0		0	0	S				0			0			0		*			*	*	*																
OUTPUT_INTERFACE			S						0		0		S							S			0		*			*	*	*																A
SUBSYSTEM			0						0		0		S										0					*	*	*																
VALIDATION:				X																														X		X					X				X	
PERFORMANCE_REQUIREMENT				S					0		0		S										0					*	*	*											*					
VALIDATION_PATH				0					0		0		S										0		*			*	*	*				*		*							*			
VALIDATION_POINT									0		0		S								S		0		*			*	*	*															A	A

X = ITEM DEFINED IN SEGMENT

S = ELEMENT MAY BE SUBJECT OF RELATIONSHIP (OR OBJECT OF COMPLEMENTARY RELATIONSHIP)

0 = ELEMENT MAY BE OBJECT OF RELATIONSHIP (OR SUBJECT OF COMPLEMENTARY RELATIONSHIP)

* = ATTRIBUTE APPLIES TO ELEMENT

A = ELEMENT HAS STRUCTURE APPLICABILITY

D.3 DEFINITIONS OF RSL ELEMENT TYPES, RELATIONSHIPS, ATTRIBUTES, AND PREDEFINED ELEMENTS

```
ELEMENT_TYPE:  ALPHA
               (* A BASIC PROCESSING STEP IN THE FUNCTIONAL
                  REQUIREMENTS. *).
   STRUCTURE APPLICABILITY:  NET.

ELEMENT_TYPE:  DATA
               (* A SINGLE PIECE OF INFORMATION OR SET OF
                  INFORMATION THAT IS EITHER REQUIRED IN THE
                  IMPLEMENTED SOFTWARE OR IS NEEDED FOR
                  DESCRIPTIVE PURPOSES. *).

ELEMENT_TYPE:  DECISION
               (* A CHOICE OR INTERPRETATION THAT HAS BEEN MADE
                  IN ORDER TO ESTABLISH FUNCTIONAL AND/OR
                  PERFORMANCE REQUIREMENTS BASED ON ONE OR MORE
                  ORIGINATING_REQUIREMENTS.  THIS MEANS THAT THE
                  LOWER LEVEL REQUIREMENTS ARE A RESULT OF
                  DERIVATION, NOT SIMPLY ALLOCATION. *).

ELEMENT_TYPE:  ENTITY_CLASS
               (* A GENERAL CATEGORY OF OBJECTS OUTSIDE THE DATA
                  PROCESSING SUBSYSTEM.  THE OBJECTS MAY BE REAL
                  OR PERCEIVED AND ARE THOSE IN THE ENVIRONMENT
                  ABOUT WHICH THE DATA PROCESSING SUBSYSTEM MUST
                  MAINTAIN INFORMATION.  FOR EXAMPLE, AN
                  ENTITY_CLASS MIGHT BE TARGET OR INTERCEPTOR.
                  WHEN THE EXISTENCE OF AN OBJECT IN AN
                  ENTITY_CLASS IS DETERMINED, FILES AND DATA MAY
                  BE TEMPORARILY CREATED TO MAINTAIN INFORMATION
                  ABOUT IT. *).

ELEMENT_TYPE:  ENTITY_TYPE
               (* A SUBSET WITHIN A GENERAL CLASS (ENTITY_CLASS)
                  OF OBJECTS OUTSIDE THE DATA PROCESSING
                  SUBSYSTEM ABOUT WHICH THE DATA PROCESSOR MUST
                  MAINTAIN INFORMATION.  FOR EXAMPLE,
                  ENTITY_TYPES WITHIN THE ENTITY_CLASS TARGET
                  MIGHT BE DETECTION, POTENTIALLY, NON-
                  THREATENING, THREATENING, ETC.  WHEN A
                  PARTICULAR OBJECT IN AN ENTITY_CLASS IS
                  DETERMINED TO BE OF A SPECIFIC TYPE, THE
                  OBJECT CAN BE SET TO THE TYPE AND DATA AND
                  FILES PERTINENT TO OBJECTS OF THAT TYPE
                  TEMPORARILY CREATED TO MAINTAIN INFORMATION
                  ABOUT THE OBJECT. *).
```

```
ELEMENT_TYPE:  EVENT
               (* AN IDENTIFIED POINT IN THE SEQUENCE OF
                  PROCESSING SPECIFIED BY ONE OR MORE R_NETS (OR
                  SUBNETS) WHICH CAUSES THE ENABLEMENT OF AN
                  R_NET. AN EVENT MAY BE USED TO SPECIFY A
                  VALIDATION_PATH. *).
   STRUCTURE APPLICABILITY:  NET.
   STRUCTURE APPLICABILITY:  PATH.

ELEMENT_TYPE:  FILE
               (* AN AGGREGATION OF INSTANCES OF DATA, EACH
                  INSTANCE OF WHICH IS TREATED IN THE SAME
                  MANNER. *).

ELEMENT_TYPE:  INPUT_INTERFACE
               (* A PORT BETWEEN THE DATA PROCESSING SUBSYSTEM
                  AND ANOTHER SUBSYSTEM (E.G., A RADAR) THROUGH
                  WHICH DATA IS PASSED TO THE DATA PROCESSING
                  SUBSYSTEM. AN INPUT_INTERFACE APPEARS AS THE
                  FIRST NODE OF ONE AND ONLY ONE R_NET
                  STRUCTURE. *).
   STRUCTURE APPLICABILITY:  NET.

ELEMENT_TYPE:  MESSAGE
               (* AN AGGREGATION OF DATA AND FILES THAT PASS
                  THROUGH AN INTERFACE AS A LOGICAL UNIT. *).

ELEMENT_TYPE:  ORIGINATING_REQUIREMENT
               (* A HIGHER LEVEL REQUIREMENT FROM WHICH LOWER
                  LEVEL REQUIREMENTS (THOSE EXPRESSED IN THE RSL)
                  ARE TRACEABLE. *).

ELEMENT_TYPE:  OUTPUT_INTERFACE
               (* A PORT BETWEEN THE DATA PROCESSING SUBSYSTEM
                  AND ANOTHER PART OF THE SYSTEM (E.G., A RADAR),
                  THROUGH WHICH DATA IS PASSED TO THE OTHER
                  SUBSYSTEM. AN OUTPUT_INTERFACE MAY APPEAR ON
                  AN R_NET OR SUBNET STRUCTURE AS THE LAST NODE
                  OF A PATH. *).
   STRUCTURE APPLICABILITY:  NET.

ELEMENT_TYPE:  PERFORMANCE_REQUIREMENT
               (* AN ANALYTIC PERFORMANCE REQUIREMENT OR
                  NON-STIMULUS-RESPONSE TIMING REQUIREMENT WHICH
                  IS TO BE MET BY THE DATA PROCESSING
                  SUBSYSTEM. *).
```

```
ELEMENT_TYPE:   R_NET
                (* THE ORDER OF LOGICAL PROCESSING THAT MUST BE
                   PERFORMED BY THE DATA PROCESSING SUBSYSTEM IN
                   RESPONSE TO EXTERNAL OR INTERNAL STIMULI.  THE
                   PROCESSING STEPS ARE ALPHAS OR SUBNETS WHICH
                   MAY BE EXPANDED TO LOWER LEVELS OF DETAIL.  IN
                   ADDITION TO PROCESSING STEPS, THE R_NET
                   STRUCTURE MAY CONTAIN INTERFACES, EVENTS,
                   VALIDATION_POINTS, ANDS, ORS, SELECTS, AND FOR
                   EACH NODES; IT MUST BE ENABLED AND
                   TERMINATED. *).

ELEMENT_TYPE:   SOURCE
                (* SOURCE OR AUXILIARY MATERIAL FOR REQUIREMENTS,
                   I.E., ORIGINATING POINT FOR ONE OR MORE
                   ORIGINATING_REQUIREMENTS, DOCUMENTATION OF
                   TRADE-OFF STUDIES, OR BACKGROUND MATERIAL FOR
                   REQUIREMENTS ELEMENTS. *).

ELEMENT_TYPE:   SUBNET
                (* A SEQUENCE OF LOGICAL PROCESSING STEPS THAT
                   MUST BE PERFORMED TO ACCOMPLISH THE
                   REQUIREMENTS OF THE NEXT HIGHER NETWORK
                   (SUBNET OR R_NET). *).
   STRUCTURE APPLICABILITY:  NET.

ELEMENT_TYPE:   SUBSYSTEM
                (* A PART OF THE SYSTEM (E.G., A RADAR) WHICH
                   COMMUNICATES WITH THE DATA PROCESSING
                   SUBSYSTEM. *).

ELEMENT_TYPE:   SYNONYM
                (* A SYNONYM IS AN ALTERNATE NAME THAT CAN BE USED
                   IN PLACE OF THE PRIME NAME OF AN ELEMENT.  IT
                   IS USED AS AN ABBREVIATION IN MOST CASES, BUT
                   MAY BE USED FOR OTHER REASONS.  NOTE:  IN THE
                   RSL DEFINITIONS OF RELATIONSHIPS AND
                   ATTRIBUTES, "ALL" ALWAYS IMPLIES "ALL EXCEPT
                   SYNONYM". *).

ELEMENT_TYPE:   UNSTRUCTURED_REQUIREMENT
                (* A REQUIREMENT THAT MUST BE PASSED TO THE
                   SOFTWARE DESIGNER BUT THAT DOES NOT FIT INTO
                   THE STRUCTURED FRAMEWORK PROVIDED BY RSL.  THIS
                   ELEMENT MIGHT BE USED BECAUSE THE
                   REQUIREMENT IN QUESTION IS TOO UNCOMMON TO
                   JUSTIFY DEFINITION OF A NEW TYPE OF ELEMENT, A
                   NEW RELATIONSHIP, OR A NEW ATTRIBUTE.  (AN
                   EXAMPLE OF AN UNSTRUCTURED_REQUIREMENT MIGHT
                   BE PRECLUSION OF USING A MULTIPROCESSOR WITH
                   ASSOCIATIVE MEMORY.) *).
```

```
ELEMENT_TYPE:  VALIDATION_PATH
               (* A PATH OF PROCESSING OVER WHICH QUANTITATIVE
                  VALIDATION TESTING WILL BE PERFORMED.  A PATH
                  IS SPECIFIED USING VALIDATION_POINTS AND
                  EVENTS AND MUST CORRESPOND TO A ROUTE THROUGH
                  AN R_NET OR THROUGH R_NETS CONNECTED BY
                  EVENTS. *).

ELEMENT_TYPE:  VALIDATION_POINT
               (* A LOGICAL POINT IN THE PROCESSING SPECIFIED BY
                  AN R_NET OR SUBNET AT WHICH DATA MUST BE
                  OBTAINABLE IN THE IMPLEMENTED SOFTWARE IN ORDER
                  TO VALIDATE THAT THE PERFORMANCE REQUIREMENTS
                  HAVE BEEN FULFILLED. *).
   STRUCTURE APPLICABILITY:  NET.
   STRUCTURE APPLICABILITY:  PATH.

ELEMENT_TYPE:  VERSION
               (* THE AGGREGATION OF REQUIREMENTS THAT ARE TO
                  APPLY AS A UNIT TO THE DATA PROCESSING
                  SUBSYSTEM AT A PARTICULAR TIME.  LOOP_1,
                  LOOP_2, ETC., ARE VERSIONS, AS IS AN IOC
                  SYSTEM. *).

RELATIONSHIP:  ASSOCIATES
               (* IDENTIFIES WHICH DATA AND FILES COME INTO
                  EXISTENCE WHEN A DATA PROCESSING STEP (AN
                  ALPHA) EITHER CREATES AN INSTANCE OF AN
                  ENTITY_CLASS OR SETS THE ENTITY_TYPE OF AN
                  INSTANCE OF AN ENTITY_CLASS.  DATA AND FILES
                  CAN BE ASSOCIATED WITH ONLY ONE ENTITY_CLASS.
                  DATA AND FILES MAY BE ASSOCIATED WITH SEVERAL
                  ENTITY_TYPES PROVIDED THE ENTITY_TYPES
                  COMPOSE THE SAME ENTITY_CLASS.  DATA AND FILES
                  THAT ARE ASSOCIATED WITH AN ENTITY_TYPE OR
                  ENTITY_CLASS MAY NOT ALSO MAKE A MESSAGE.  DATA
                  THAT IS ASSOCIATED WITH AN ENTITY_TYPE OR
                  ENTITY_CLASS MAY NOT ALSO BE CONTAINED IN A
                  FILE. *).
   COMPLEMENTARY RELATIONSHIP:  ASSOCIATED ("WITH").
   SUBJECT ELEMENT_TYPE:  ENTITY_CLASS
                          ENTITY_TYPE.
   OBJECT ELEMENT_TYPE:  DATA
                         FILE.

RELATIONSHIP:  COMPOSES
               (* IDENTIFIES TO WHICH ENTITY_CLASS AN ENTITY_TYPE
                  BELONGS.  AN ENTITY_TYPE COMPOSES ONLY ONE
                  ENTITY_CLASS; AN ENTITY_CLASS IS COMPOSED OF AT
                  LEAST ONE ENTITY_TYPE. *).
   COMPLEMENTARY RELATIONSHIP:  COMPOSED ("OF").
   SUBJECT ELEMENT_TYPE:  ENTITY_TYPE.
   OBJECT ELEMENT_TYPE:  ENTITY_CLASS.
```

```
RELATIONSHIP:  CONNECTS ("TO")
               (* IDENTIFIES WITH WHICH SUBSYSTEM THE
                  INPUT_INTERFACE OR OUTPUT_INTERFACE
                  COMMUNICATES.  AN INTERFACE CONNECTS TO ONLY
                  ONE SUBSYSTEM. *).
   COMPLEMENTARY RELATIONSHIP:  CONNECTED ("TO").
   SUBJECT ELEMENT_TYPE:  INPUT_INTERFACE
                          OUTPUT_INTERFACE.
   OBJECT ELEMENT_TYPE:  SUBSYSTEM.

RELATIONSHIP:  CONSTRAINS
               (* IDENTIFIES TO WHICH VALIDATION_PATH(S) THE
                  PERFORMANCE_REQUIREMENT APPLIES. *).
   COMPLEMENTARY RELATIONSHIP:  CONSTRAINED ("BY").
   SUBJECT ELEMENT_TYPE:  PERFORMANCE_REQUIREMENT.
   OBJECT ELEMENT_TYPE:  VALIDATION_PATH.

RELATIONSHIP:  CONTAINS
               (* IDENTIFIES THE MEMBERS OF EACH INSTANCE IN A
                  FILE.  DATA MAY BE CONTAINED IN ONLY ONE FILE.
                  DATA THAT IS CONTAINED IN A FILE MAY NOT ALSO
                  MAKE A MESSAGE NOR MAY IT BE ASSOCIATED WITH AN
                  ENTITY_CLASS OR ENTITY_TYPE. *).
   COMPLEMENTARY RELATIONSHIP:  CONTAINED ("IN").
   SUBJECT ELEMENT_TYPE:  FILE.
   OBJECT ELEMENT_TYPE:  DATA.

RELATIONSHIP:  CREATES
               (* INDICATES THAT THE ALPHA CREATES AN INSTANCE OF
                  THE ENTITY_CLASS(ES).  CREATION OF AN ENTITY
                  INSTANCE IN A CLASS OCCURS IMMEDIATELY AT THE
                  BEGINNING OF AN ALPHA WHICH CREATES THE
                  ENTITY_CLASS.  ONLY ONE NEW ENTITY INSTANCE IS
                  CREATED. *).
   COMPLEMENTARY RELATIONSHIP:  CREATED ("BY").
   SUBJECT ELEMENT_TYPE:  ALPHA.
   OBJECT ELEMENT_TYPE:  ENTITY_CLASS.

RELATIONSHIP:  DELAYS
               (* THE ENABLEMENT OF R_NETS BY THE EVENT IS
                  POSTPONED FOR THE AMOUNT OF TIME SPECIFIED IN
                  THE DATA.  ONLY ONE DATA MAY DELAY AN EVENT;
                  THIS DATA MUST NOT INCLUDE OTHER DATA.  FOR
                  SIMULATION PURPOSES, THE VALUE OF THIS DATA
                  MUST BE IN UNITS OF SECONDS. *).
   COMPLEMENTARY RELATIONSHIP:  DELAYED ("BY").
   SUBJECT ELEMENT_TYPE:  DATA.
   OBJECT ELEMENT_TYPE:  EVENT.
```

```
RELATIONSHIP:  DESTROYS
               (* INDICATES THAT THE ALPHA DESTROYS AN INSTANCE
                  (THE CURRENTLY SELECTED ONE) OF THE
                  ENTITY_CLASS(ES).  IDENTIFICATION OF THE
                  INSTANCE IS PERFORMED BY A SELECT OR FOR EACH
                  NODE ON A NETWORK.  DESTRUCTION OF THE INSTANCE
                  OCCURS IMMEDIATELY BEFORE COMPLETION OF
                  PROCESSING IN THE ALPHA. *).
   COMPLEMENTARY RELATIONSHIP:  DESTROYED ("BY").
   SUBJECT ELEMENT_TYPE:  ALPHA.
   OBJECT ELEMENT_TYPE:  ENTITY_CLASS.

RELATIONSHIP:  DOCUMENTS
               (* THE SOURCE MATERIAL PROVIDES AUXILIARY
                  INFORMATION ABOUT OR IS THE ORIGINATING POINT
                  FOR THE OBJECT ELEMENT. *).
   COMPLEMENTARY RELATIONSHIP:  DOCUMENTED ("BY").
   SUBJECT ELEMENT_TYPE:  SOURCE.
   OBJECT ELEMENT_TYPE:  ALPHA
                         DATA
                         DECISION
                         ENTITY_CLASS
                         ENTITY_TYPE
                         EVENT
                         FILE
                         INPUT_INTERFACE
                         MESSAGE
                         ORIGINATING_REQUIREMENT
                         OUTPUT_INTERFACE
                         PERFORMANCE_REQUIREMENT
                         R_NET
                         SUBNET
                         SUBSYSTEM
                         UNSTRUCTURED_REQUIREMENT
                         VALIDATION_PATH
                         VALIDATION_POINT
                         VERSION.

RELATIONSHIP:  ENABLES
               (* INDICATES THAT WHEN THE PROCESSING CONTROL FLOW
                  PASSES THROUGH THE EVENT ON AN R_NET, OR WHEN
                  DATA IS AVAILABLE AT THE INPUT_INTERFACE, THE
                  FUNCTIONAL PROCESSING SPECIFIED BY THE R_NET
                  CAN BE BEGUN.  AN R_NET MUST BE ENABLED AND CAN
                  BE ENABLED EITHER BY ONE AND ONLY ONE
                  INPUT_INTERFACE OR BY ONE OR MORE EVENTS. *).
   COMPLEMENTARY RELATIONSHIP:  ENABLED ("BY").
   SUBJECT ELEMENT_TYPE:  EVENT
                          INPUT_INTERFACE.
   OBJECT ELEMENT_TYPE:  R_NET.
```

```
RELATIONSHIP:  EQUATES ("TO")
               (* DEFINES AN ALTERNATE NAME FOR AN ELEMENT.  THE
                  OBJECT OF EQUATES IS CALLED THE PRIME NAME.
                  THE SUBJECT NAME CAN BE USED FOR INPUT TO THE
                  ASSM, BUT ALL RELATIONSHIPS, ATTRIBUTES, AND
                  STRUCTURES SO DEFINED ARE ACTUALLY
                  CHARACTERISTICS OF THE PRIME NAME. *).
   COMPLEMENTARY RELATIONSHIP:  EQUATED ("TO").
   SUBJECT ELEMENT_TYPE:  SYNONYM.
   OBJECT ELEMENT_TYPE:  ALPHA
                         DATA
                         DECISION
                         ENTITY_CLASS
                         ENTITY_TYPE
                         EVENT
                         FILE
                         INPUT_INTERFACE
                         MESSAGE
                         ORIGINATING_REQUIREMENT
                         OUTPUT_INTERFACE
                         PERFORMANCE_REQUIREMENT
                         R_NET
                         SOURCE
                         SUBNET
                         SUBSYSTEM
                         UNSTRUCTURED_REQUIREMENT
                         VALIDATION_PATH
                         VALIDATION_POINT
                         VERSION.

RELATIONSHIP:  FORMS
               (* INDICATES THAT THE ALPHA ESTABLISHES THE
                  MESSAGE AS THE ONE TO BE PASSED BY THE
                  CORRESPONDING OUTPUT_INTERFACE (THE
                  OUTPUT_INTERFACE WHICH PASSES THE MESSAGE)
                  WHEN THAT INTERFACE IS ENCOUNTERED ON THE NET.
                  AN ALPHA MAY FORM SEVERAL MESSAGES PROVIDED
                  THEY ARE PASSED BY DIFFERENT
                  OUTPUT_INTERFACES. *).
   COMPLEMENTARY RELATIONSHIP:  FORMED ("BY").
   SUBJECT ELEMENT_TYPE:  ALPHA.
   OBJECT ELEMENT_TYPE:  MESSAGE.
```

```
RELATIONSHIP:  IMPLEMENTS
               (* DEFINES THE VERSION(S) TO WHICH THE ELEMENT
                  APPLIES. *).
   COMPLEMENTARY RELATIONSHIP:  IMPLEMENTED ("BY").
   SUBJECT ELEMENT_TYPE:  ALPHA
                          DATA
                          DECISION
                          ENTITY_CLASS
                          ENTITY_TYPE
                          EVENT
                          FILE
                          INPUT_INTERFACE
                          MESSAGE
                          ORIGINATING_REQUIREMENT
                          OUTPUT_INTERFACE
                          PERFORMANCE_REQUIREMENT
                          R_NET
                          SUBNET
                          SUBSYSTEM
                          UNSTRUCTURED_REQUIREMENT
                          VALIDATION_PATH
                          VALIDATION_POINT.
   OBJECT ELEMENT_TYPE:  VERSION.

RELATIONSHIP:  INCLUDES
               (* INDICATES A HIERARCHICAL RELATIONSHIP BETWEEN
                  DATA.  IF A INCLUDES B, THEN OBTAINING A WILL
                  OBTAIN B. *).
   COMPLEMENTARY RELATIONSHIP:  INCLUDED ("IN").
   SUBJECT ELEMENT_TYPE:  DATA.
   OBJECT ELEMENT_TYPE:  DATA.

RELATIONSHIP:  INCORPORATES
               (* INDICATES A HIERARCHICAL RELATIONSHIP BETWEEN
                  ORIGINATING_REQUIREMENTS.  THE SCOPE OF THE
                  SUBJECT (HIGHER LEVEL) ORIGINATING_REQUIREMENT
                  INCLUDES THE OBJECT (LOWER LEVEL)
                  ORIGINATING_REQUIREMENT. *).
   COMPLEMENTARY RELATIONSHIP:  INCORPORATED ("IN").
   SUBJECT ELEMENT_TYPE:  ORIGINATING_REQUIREMENT.
   OBJECT ELEMENT_TYPE:  ORIGINATING_REQUIREMENT.

RELATIONSHIP:  INPUTS
               (* IDENTIFIES THE DATA AND FILES USED BY THE
                  ALPHA. *).
   COMPLEMENTARY RELATIONSHIP:  INPUT ("TO").
   SUBJECT ELEMENT_TYPE:  ALPHA.
   OBJECT ELEMENT_TYPE:  DATA
                         FILE.
```

```
RELATIONSHIP:  MAKES
               (* INDICATES THAT THE DATA OR FILE IS A LOGICAL
                  COMPONENT OF THE MESSAGE.  A DATA OR FILE MAY
                  MAKE SEVERAL MESSAGES.  DATA AND FILES THAT
                  MAKE A MESSAGE MAY NOT ALSO BE ASSOCIATED WITH
                  AN ENTITY_TYPE OR ENTITY_CLASS.  DATA THAT
                  MAKES A MESSAGE MAY NOT ALSO BE CONTAINED IN A
                  FILE. *).
   COMPLEMENTARY RELATIONSHIP:  MADE ("BY").
   SUBJECT ELEMENT_TYPE:  DATA
                          FILE.
   OBJECT ELEMENT_TYPE:  MESSAGE.

RELATIONSHIP:  ORDERS
               (* INDICATES THAT THE VALUE OF THE DATA IS USED TO
                  ORDER THE INSTANCES OF THE FILE.  A FILE MAY BE
                  ORDERED BY ONLY ONE DATA; THE DATA MAY NOT
                  INCLUDE OTHER DATA AND SHOULD BE CONTAINED IN
                  THE FILE. *).
   COMPLEMENTARY RELATIONSHIP:  ORDERED ("BY").
   SUBJECT ELEMENT_TYPE:  DATA.
   OBJECT ELEMENT_TYPE:  FILE.

RELATIONSHIP:  OUTPUTS
               (* IDENTIFIES THE DATA AND FILES WHOSE VALUES OR
                  CONTENTS ARE MODIFIED BY THE ALPHA. *).
   COMPLEMENTARY RELATIONSHIP:  OUTPUT ("FROM").
   SUBJECT ELEMENT_TYPE:  ALPHA.
   OBJECT ELEMENT_TYPE:  DATA
                         FILE.

RELATIONSHIP:  PASSES
               (* IDENTIFIES THE LOGICAL UNITS OF INFORMATION
                  WHICH ARE PASSED THROUGH THE INTERFACE.  AN
                  INTERFACE MAY PASS SEVERAL MESSAGES; A GIVEN
                  MESSAGE MAY BE PASSED THROUGH ONLY ONE
                  INTERFACE. *).
   COMPLEMENTARY RELATIONSHIP:  PASSED ("THROUGH").
   SUBJECT ELEMENT_TYPE:  INPUT_INTERFACE
                          OUTPUT_INTERFACE.
   OBJECT ELEMENT_TYPE:  MESSAGE.

RELATIONSHIP:  RECORDS
               (* IDENTIFIES THE PARTICULAR DATA AND FILES WHICH
                  ARE TO BE MADE AVAILABLE AT THE
                  VALIDATION_POINT FOR PERFORMANCE
                  EVALUATION. *).
   COMPLEMENTARY RELATIONSHIP:  RECORDED ("BY").
   SUBJECT ELEMENT_TYPE:  VALIDATION_POINT.
   OBJECT ELEMENT_TYPE:  DATA
                         FILE.
```

```
RELATIONSHIP:  SETS
               (* INDICATES THAT THE ALPHA ESTABLISHES AN
                  INSTANCE (THE CURRENTLY SELECTED ONE) OF AN
                  ENTITY_CLASS TO BE OF THE ENTITY_TYPE.
                  IDENTIFICATION OF THE INSTANCE IS PERFORMED BY
                  A SELECT OR FOR EACH NODE ON A NETWORK.  AN
                  ALPHA MAY SET SEVERAL ENTITY_TYPES PROVIDED THE
                  ENTITY_TYPES DO NOT COMPOSE THE SAME
                  ENTITY_CLASS.  THE SETTING OF AN ENTITY_TYPE
                  OCCURS IMMEDIATELY IN AN ALPHA SUBSEQUENT TO
                  ANY ENTITY CREATIONS. *).
   COMPLEMENTARY RELATIONSHIP:  SET ("BY").
   SUBJECT ELEMENT_TYPE:  ALPHA.
   OBJECT ELEMENT_TYPE:  ENTITY_TYPE.

RELATIONSHIP:  TRACES ("TO")
               (* IDENTIFIES THE ELEMENTS (LOWER LEVEL
                  REQUIREMENTS) TO OR FROM WHICH THE HIGHER
                  LEVEL REQUIREMENT (ORIGINATING_REQUIREMENT OR
                  DECISION) HAVE BEEN ALLOCATED OR DERIVED. *).
   COMPLEMENTARY RELATIONSHIP:  TRACED ("FROM").
   SUBJECT ELEMENT_TYPE:  DECISION
                          ORIGINATING_REQUIREMENT.
   OBJECT ELEMENT_TYPE:  ALPHA
                         DATA
                         DECISION
                         ENTITY_CLASS
                         ENTITY_TYPE
                         EVENT
                         FILE
                         INPUT_INTERFACE
                         MESSAGE
                         OUTPUT_INTERFACE
                         PERFORMANCE_REQUIREMENT
                         R_NET
                         SUBNET
                         SUBSYSTEM
                         UNSTRUCTURED_REQUIREMENT
                         VALIDATION_PATH
                         VALIDATION_POINT
                         VERSION.

ATTRIBUTE:  ALTERNATIVES
            (* THE ALTERNATIVES THAT HAVE BEEN CONSIDERED TO
               RESOLVE A PROBLEM RESULTING IN A DECISION. *).
   APPLICABLE ELEMENT_TYPE:  DECISION.
   VALUE:  TEXT.
```

```
ATTRIBUTE: ARTIFICIALITY
           (* THE DEGREE OF FLEXIBILITY ALLOWED IN IMPLEMENTING
              THE ELEMENT IN THE SOFTWARE. *).
   APPLICABLE ELEMENT_TYPE:  ALPHA
                             DATA
                             ENTITY_CLASS
                             ENTITY_TYPE
                             EVENT
                             FILE
                             INPUT_INTERFACE
                             MESSAGE
                             OUTPUT_INTERFACE
                             R_NET
                             SUBNET
                             VALIDATION_PATH
                             VALIDATION_POINT.
   VALUE: ARTIFICIAL
          (* THE ELEMENT HAS BEEN DEFINED FOR EXPLANATORY OR
             SIMULATION PURPOSES IN THE REQUIREMENTS STATEMENT
             AND NEED NOT BE PRESENT IN THE SOFTWARE. *).
   VALUE: VALIDATION
          (* THE ELEMENT IS NECESSARY FOR PERFORMANCE
             REQUIREMENTS EVALUATION BUT IS NOT REQUIRED IN THE
             OPERATIONAL SOFTWARE. *).
   VALUE: IMPLEMENT_PRECISELY
          (* THE ELEMENT MUST BE IMMPLEMENTED IN THE SOFTWARE
             EXACTLY AS DEFINED. *).
   VALUE: IMPLEMENT_APPROXIMATELY
          (* THE ELEMENT MUST BE IMPLEMENTED IN THE SOFTWARE,
             BUT THE PRECISE IMPLEMENTATION IS LEFT TO THE
             PROCESS DESIGNER. *).

ATTRIBUTE: BETA
           (* THE PROCEDURAL CODE (PASCAL) FOR FUNCTIONALLY
              MODELING THE PROCESSING STEP. THE CODE IS NOT
              PROCESSED BY THE RSL TRANSLATOR BUT IS PROCESSED
              BY THE SIMULATION GENERATION FUNCTION AND THE
              COMPILER. A BETA MAY USE THE SPECIAL CREATE,
              DESTROY, SELECT AND FOR EACH OPERATIONS ON
              FILES. *).
   APPLICABLE ELEMENT_TYPE:  ALPHA.
   VALUE: TEXT.

ATTRIBUTE: CHOICE
           (* THE ALTERNATIVE SELECTED TO SOLVE A PROBLEM
              LEADING TO A DECISION. THE RATIONALE FOR THE
              CHOICE SHOULD BE INCLUDED HERE. *).
   APPLICABLE ELEMENT_TYPE:  DECISION.
   VALUE: TEXT.
```

```
ATTRIBUTE: COMPLETENESS
       (* THE DEGREE TO WHICH THE DEFINITION OF AN ELEMENT IS
          IN FINAL FORM. *).
   APPLICABLE ELEMENT_TYPE:  ALPHA
                             DATA
                             DECISION
                             ENTITY_CLASS
                             ENTITY_TYPE
                             EVENT
                             FILE
                             INPUT_INTERFACE
                             MESSAGE
                             ORIGINATING_REQUIREMENT
                             OUTPUT_INTERFACE
                             PERFORMANCE_REQUIREMENT
                             R_NET
                             SOURCE
                             SUBNET
                             SUBSYSTEM
                             UNSTRUCTURED_REQUIREMENT
                             VALIDATION_PATH
                             VALIDATION_POINT
                             VERSION.
   VALUE: CHANGEABLE
       (* ALTHOUGH ALL RELATIONSHIPS, ATTRIBUTES, AND
          STRUCTURES MAY BE DEFINED FOR THE ELEMENT, SOME OF
          THEM WILL PROBABLY BE CHANGED.  INFORMATION ABOUT
          THE ELEMENT IS BELIEVED TO BE CORRECT, BUT IS
          SUBJECT TO CHANGE. *).
   VALUE: INCOMPLETE
       (* THE DEFINITION OF THE ELEMENT IS KNOWN TO BE
          INCOMPLETE.  THEREFORE, EVEN IF RELATIONSHIPS,
          ATTRIBUTES, AND STRUCTURES ARE STATED, THE ELEMENT
          DEFINITION IS STILL INCOMPLETE. *).
   VALUE: COMPLETE
       (* THE DEFINITION OF THE ELEMENT SHOULD BE ASSUMED TO
          BE COMPLETE AND WILL PROBABLY NOT CHANGE. *).
```

```
ATTRIBUTE: DESCRIPTION
           (* ANY FREE FORM TEXTUAL MATERIAL DESCRIBING THE
              ELEMENT. *).
   APPLICABLE ELEMENT_TYPE: ALPHA
                            DATA
                            DECISION
                            ENTITY_CLASS
                            ENTITY_TYPE
                            EVENT
                            FILE
                            INPUT_INTERFACE
                            MESSAGE
                            ORIGINATING_REQUIREMENT
                            OUTPUT_INTERFACE
                            PERFORMANCE_REQUIREMENT
                            R_NET
                            SOURCE
                            SUBNET
                            SUBSYSTEM
                            UNSTRUCTURED_REQUIREMENT
                            VALIDATION_PATH
                            VALIDATION_POINT
                            VERSION.
   VALUE: TEXT.

ATTRIBUTE: ENTERED_BY
           (* THE IDENTITY OF THE LAST PERSON TO ENTER
              INFORMATION ABOUT THE ELEMENT. *).
   APPLICABLE ELEMENT_TYPE: ALPHA
                            DATA
                            DECISION
                            ENTITY_CLASS
                            ENTITY_TYPE
                            EVENT
                            FILE
                            INPUT_INTERFACE
                            MESSAGE
                            ORIGINATING_REQUIREMENT
                            OUTPUT_INTERFACE
                            PERFORMANCE_REQUIREMENT
                            R_NET
                            SOURCE
                            SUBNET
                            SUBSYSTEM
                            UNSTRUCTURED_REQUIREMENT
                            VALIDATION_PATH
                            VALIDATION_POINT
                            VERSION.
   VALUE: TEXT.
```

```
ATTRIBUTE:  GAMMA
            (* THE PROCEDURAL CODE (PASCAL) FOR ANALYTICALLY
               MODELING A PROCESSING STEP.  THE CODE IS NOT
               PROCESSED BY THE RSL TRANSLATOR BUT IS PROCESSED BY
               THE SIMULATION GENERATION FUNCTION AND THE
               COMPILER.  A GAMMA MAY USE THE SPECIAL CREATE,
               DESTROY, SELECT AND FOR EACH OPERATIONS ON
               FILES. *).
   APPLICABLE ELEMENT_TYPE:  ALPHA.
   VALUE: TEXT.

ATTRIBUTE:  INITIAL_VALUE
            (* THE INITIAL VALUE A DATA ITEM IS REQUIRED TO HAVE
               IN THE IMPLEMENTED SOFTWARE.  THIS VALUE WILL BE
               ASSUMED BY THE DATA ITEM WHEN IT COMES INTO
               EXISTENCE IN A SIMULATION. *).
   APPLICABLE ELEMENT_TYPE:  DATA.
   VALUE: NAMED.
   VALUE: NUMERIC.

ATTRIBUTE:  LOCALITY
            (* THE ACCESSIBILITY AND LIFETIME OF A DATA OR
               FILE. *).
   APPLICABLE ELEMENT_TYPE:  DATA
                             FILE.
   VALUE: GLOBAL
         (* GLOBAL DATA AND FILES ARE ACCESSIBLE BY MORE THAN
            ONE R_NET AND MAY EXIST THROUGHOUT EXECUTION OF THE
            SYSTEM.  DATA AND FILES WHICH ARE ASSOCIATED WITH AN
            ENTITY_TYPE OR AN ENTITY_CLASS ARE BY DEFINITION
            GLOBAL. *).
   VALUE: LOCAL
         (* LOCAL DATA AND FILES ARE ASSOCIATED WITH THE R_NETS
            IN WHICH THEY ARE USED AND EXIST ONLY DURING THE
            INVOCATION OF THE R_NET TO WHICH THEY ARE LOCAL.
            DATA AND FILES WHICH MAKE A MESSAGE ARE BY
            DEFINITION LOCAL. *).

ATTRIBUTE:  MAXIMUM_TIME
            (* THE MAXIMUM TIME THAT CAN BE TAKEN TO TRAVERSE THE
               VALIDATION_PATH.  THE TIME IS SPECIFIED IN THE UNITS
               STATED IN THE UNITS ATTRIBUTE. *).
   APPLICABLE ELEMENT_TYPE:  VALIDATION_PATH.
   VALUE: NUMERIC.

ATTRIBUTE:  MAXIMUM_VALUE
            (* THE MAXIMUM VALUE A DATA ITEM MAY ASSUME.  THE
               VALUE IS IN THE UNITS STATED IN THE UNITS
               ATTRIBUTE AND SHOULD BE CONSISTENT WITH THE TYPE OF
               THE DATA. *).
   APPLICABLE ELEMENT_TYPE:  DATA.
   VALUE: NUMERIC.
```

```
ATTRIBUTE:  MINIMUM_TIME
            (* THE MINIMUM TIME THAT CAN BE TAKEN TO TRAVERSE THE
               VALIDATION_PATH.  THE TIME IS SPECIFIED IN THE
               UNITS DESIGNATED BY THE UNITS ATTRIBUTE. *).
   APPLICABLE ELEMENT_TYPE:  VALIDATION_PATH.
   VALUE: NUMERIC.

ATTRIBUTE:  MINIMUM_VALUE
            (* THE MINIMUM VALUE A DATA ITEM MAY ASSUME.  THE
               VALUE IS IN THE UNITS STATED IN THE UNITS
               ATTRIBUTE AND SHOULD BE CONSISTENT WITH THE TYPE OF
               THE DATA. *).
   APPLICABLE ELEMENT_TYPE:  DATA.
   VALUE: NUMERIC.

ATTRIBUTE:  PROBLEM
            (* THE PROBLEM THAT HAS LED TO THE NEED FOR A
               DECISION. *).
   APPLICABLE ELEMENT_TYPE:  DECISION.
   VALUE: TEXT.

ATTRIBUTE:  RANGE
            (* THE NAMED VALUES THAT CAN BE ASSUMED BY A DATA WITH
               TYPE ENUMERATION. *).
   APPLICABLE ELEMENT_TYPE:  DATA.
   VALUE: TEXT
   (* THE ALLOWED VALUES ARE SEPARATED BY COMMAS. *).

ATTRIBUTE:  RESOLUTION
            (* DESCRIBES THE REQUIRED MAXIMUM VALUE OF THE LEAST
               SIGNIFICANT BIT FOR THE DATA IN UNITS SPECIFIED IN
               THE UNITS ATTRIBUTE. *).
   APPLICABLE ELEMENT_TYPE:  DATA.
   VALUE: NUMERIC.

ATTRIBUTE:  TEST
            (* PROCEDURAL CODE (PASCAL) WHICH DEFINES THE
               COMPUTATIONS NECESSARY TO TEST THE SATISFACTION OF
               A PERFORMANCE_REQUIREMENT USING DATA RECORDED BY
               VALIDATION_POINTS.  THE CODE IS NOT PROCESSED
               BY THE RSL TRANSLATOR BUT IS PROCESSED BY THE
               SIMULATION GENERATION FUNCTION AND THE COMPILER.
               A TEST CONTAINS SPECIAL RETRIEVE AND FOR EACH
               OPERATIONS TO IDENTIFY VALIDATION_POINT RECORDINGS
               AND MAY USE SELECT AND FOR EACH OPERATIONS TO
               ACCESS RECORDED FILES. *).
   APPLICABLE ELEMENT_TYPE:  PERFORMANCE_REQUIREMENT.
   VALUE: TEXT.
```

```
ATTRIBUTE:  TYPE
            (* THE TYPE FOR A DATA ITEM WHICH IS EITHER
               REFERENCED ON AN R_NET OR SUBNET OR IS USED IN A
               BETA OR GAMMA SIMULATION. *).
   APPLICABLE ELEMENT_TYPE:  DATA.
   VALUE: REAL.
   VALUE: ENUMERATION
          (* THE DATA ITEM CAN ASSUME ONLY CERTAIN VALUES
             WHICH ARE NAMES.  THE ALLOWED VALUES FOR THE DATA
             ITEM ARE SPECIFIED IN THE RANGE ATTRIBUTE. *).
   VALUE: BOOLEAN.
   VALUE: INTEGER.

ATTRIBUTE:  UNITS
            (* THE ENGINEERING UNITS OF THE VALUE OF A DATA ITEM
               OR THE UNITS IN WHICH THE MAXIMUM_TIME AND/OR
               MINIMUM_TIME FOR A VALIDATION_PATH ARE
               SPECIFIED. *).
   APPLICABLE ELEMENT_TYPE:  DATA
                             VALIDATION_PATH.
   VALUE: NAMED
          (* FOR INDIVIDUAL PROJECTS IT MAY BE DESIRABLE TO
             RESTRICT THE UNITS WHICH CAN BE USED.  IN THAT CASE,
             NAMED SHOULD BE REPLACED BY THE SPECIFIC LEGAL
             VALUE NAMES. *).

ATTRIBUTE:  USE
            (* QUALIFIES THE USE OF A DATA ITEM IN
               SIMULATION. *).
   APPLICABLE ELEMENT_TYPE:  DATA.
   VALUE: BETA
          (* THE DATA ITEM IS TO BE USED IN FUNCTIONAL
             SIMULATIONS ONLY. *).
   VALUE: GAMMA
          (* THE DATA ITEM IS TO APPEAR IN ANALYTIC
             SIMULATIONS ONLY. *).
   VALUE: BOTH
          (* THE DATA ITEM IS TO BE USED IN BOTH FUNCTIONAL AND
             ANALYTIC SIMULATIONS. *).

DATA:  CLOCK_TIME.
    DESCRIPTION:
            "A PREDEFINED DATA ITEM WHICH IS INCREMENTED AT
            THE SAME RATE AS ENGAGEMENT TIME.  EXCEPT FOR ITS
            INITIAL_VALUE WHICH IS ARBITRARY, CLOCK_TIME MAY BE
            REGARDED AS ENGAGEMENT TIME.  IT HAS NO CLOCK
            ERROR.".
    LOCALITY: GLOBAL.
    TYPE: REAL.
    UNITS: SECONDS.
    USE: BOTH.
```

```
DATA: FOUND.
     DESCRIPTION:
               "A PREDEFINED DATA ITEM WHICH IS SET TO EITHER
               TRUE OR FALSE AFTER EACH SELECT ON AN ENTITY_TYPE
               OR ENTITY_CLASS.  FOUND IS SET TO TRUE IF AN
               INSTANCE SATISFYING THE SELECTION CRITERION IS
               LOCATED; OTHERWISE, FOUND IS ASSIGNED THE VALUE
               FALSE.".
     INITIAL_VALUE: FALSE.
     LOCALITY: LOCAL.
     TYPE: BOOLEAN.
     USE: BOTH.

DATA: RECORD_FOUND.
     DESCRIPTION:
               "A PREDEFINED DATA ITEM WHICH IS SET TO EITHER TRUE
               OR FALSE AFTER EACH SELECT ON A FILE IN A BETA OR
               GAMMA.  RECORD_FOUND IS SET TO TRUE IF A RECORD
               SATISFYING THE SELECTION CRITERION IS LOCATED;
               OTHERWISE, RECORD_FOUND IS ASSIGNED THE VALUE
               FALSE.".
     INITIAL_VALUE: FALSE.
     LOCALITY: LOCAL.
     TYPE: BOOLEAN.
     USE: BOTH.
```

D.4 SUMMARY OF RSL RELATIONSHIPS AND ATTRIBUTES BY ELEMENT TYPE

```
ELEMENT_TYPE:  ALPHA
   LEGAL RELATIONSHIPS:
     CREATES:
          ENTITY_CLASS
     DESTROYS:
          ENTITY_CLASS
     FORMS:
          MESSAGE
     IMPLEMENTS:
          VERSION
     INPUTS:
          DATA
          FILE
     OUTPUTS:
          DATA
          FILE
     SETS:
          ENTITY_TYPE
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     ARTIFICIALITY:
          ARTIFICIAL
          VALIDATION
          IMPLEMENT_APPROXIMATELY
          IMPLEMENT_PRECISELY
     BETA:
          TEXT
     COMPLETENESS:
          INCOMPLETE
          COMPLETE
          CHANGEABLE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
     GAMMA:
          TEXT
```

```
ELEMENT_TYPE:  DATA
   LEGAL RELATIONSHIPS:                          LEGAL ATTRIBUTES:
      DELAYS:                                       ARTIFICIALITY:
           EVENT                                         ARTIFICIAL
      IMPLEMENTS:                                        VALIDATION
           VERSION                                       IMPLEMENT_APPROXIMATELY
      INCLUDES:                                          IMPLEMENT_PRECISELY
           DATA                                     COMPLETENESS:
      MAKES:                                             INCOMPLETE
           MESSAGE                                       COMPLETE
      ORDERS:                                            CHANGEABLE
           FILE                                     DESCRIPTION:
      ASSOCIATED ("WITH"):                               TEXT
           ENTITY_CLASS                             ENTERED_BY:
           ENTITY_TYPE                                   TEXT
      CONTAINED ("IN"):                             INITIAL_VALUE:
           FILE                                          NAMED
      DOCUMENTED ("BY"):                                 NUMERIC
           SOURCE                                   LOCALITY:
      EQUATED ("TO"):                                    GLOBAL
           SYNONYM                                       LOCAL
      INCLUDED ("IN"):                              MAXIMUM_VALUE:
           DATA                                          NUMERIC
      INPUT ("TO"):                                 MINIMUM_VALUE:
           ALPHA                                         NUMERIC
      OUTPUT ("FROM"):                              RANGE:
           ALPHA                                         TEXT
      RECORDED ("BY"):                              RESOLUTION:
           VALIDATION_POINT                              NUMERIC
      TRACED ("FROM"):                              TYPE:
           DECISION                                      REAL
           ORIGINATING_REQUIREMENT                       ENUMERATION
                                                         BOOLEAN
                                                         INTEGER
                                                    UNITS:
                                                         NAMED
                                                    USE:
                                                         BETA
                                                         BOTH
                                                         GAMMA
```

```
ELEMENT_TYPE: DECISION
   LEGAL RELATIONSHIPS:
     IMPLEMENTS:
          VERSION
     TRACES ("TO"):
          ALPHA
          DATA
          DECISION
          ENTITY_CLASS
          ENTITY_TYPE
          EVENT
          FILE
          INPUT_INTERFACE
          MESSAGE
          OUTPUT_INTERFACE
          PERFORMANCE_REQUIREMENT
          R_NET
          SUBNET
          SUBSYSTEM
          UNSTRUCTURED_REQUIREMENT
          VALIDATION_PATH
          VALIDATION_POINT
          VERSION
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     ALTERNATIVES:
          TEXT
     CHOICE:
          TEXT
     COMPLETENESS:
          CHANGEABLE
          INCOMPLETE
          COMPLETE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
     PROBLEM:
          TEXT
```

```
ELEMENT_TYPE:  ENTITY_CLASS
   LEGAL RELATIONSHIPS:
     ASSOCIATES:
          DATA
          FILE
     IMPLEMENTS:
          VERSION
     COMPOSED ("OF"):
          ENTITY_TYPE
     CREATED ("BY"):
          ALPHA
     DESTROYED ("BY"):
          ALPHA
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     ARTIFICIALITY:
          ARTIFICIAL
          VALIDATION
          IMPLEMENT_APPROXIMATELY
          IMPLEMENT_PRECISELY
     COMPLETENESS:
          INCOMPLETE
          COMPLETE
          CHANGEABLE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
```

```
ELEMENT_TYPE:  ENTITY_TYPE
   LEGAL RELATIONSHIPS:
     ASSOCIATES:
          DATA
          FILE
     COMPOSES:
          ENTITY_CLASS
     IMPLEMENTS:
          VERSION
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     SET ("BY"):
          ALPHA
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     ARTIFICIALITY:
          ARTIFICIAL
          VALIDATION
          IMPLEMENT_APPROXIMATELY
          IMPLEMENT_PRECISELY
     COMPLETENESS:
          INCOMPLETE
          COMPLETE
          CHANGEABLE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
```

```
ELEMENT_TYPE:  EVENT
   LEGAL RELATIONSHIPS:
     ENABLES:
          R_NET
     IMPLEMENTS:
          VERSION
     DELAYED ("BY"):
          DATA
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     ARTIFICIALITY:
          ARTIFICIAL
          VALIDATION
          IMPLEMENT_APPROXIMATELY
          IMPLEMENT_PRECISELY
     COMPLETENESS:
          INCOMPLETE
          COMPLETE
          CHANGEABLE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
```

```
ELEMENT_TYPE: FILE
   LEGAL RELATIONSHIPS:
     CONTAINS:
          DATA
     IMPLEMENTS:
          VERSION
     MAKES:
          MESSAGE
     ASSOCIATED ("WITH"):
          ENTITY_CLASS
          ENTITY_TYPE
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     INPUT ("TO"):
          ALPHA
     ORDERED ("BY"):
          DATA
     OUTPUT ("FROM"):
          ALPHA
     RECORDED ("BY"):
          VALIDATION_POINT
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     ARTIFICIALITY:
          ARTIFICIAL
          VALIDATION
          IMPLEMENT_APPROXIMATELY
          IMPLEMENT_PRECISELY
     COMPLETENESS:
          INCOMPLETE
          COMPLETE
          CHANGEABLE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
     LOCALITY:
          GLOBAL
          LOCAL
```

```
ELEMENT_TYPE:  INPUT_INTERFACE
   LEGAL RELATIONSHIPS:
      CONNECTS ("TO"):
             SUBSYSTEM
      ENABLES:
             R_NET
      IMPLEMENTS:
             VERSION
      PASSES:
             MESSAGE
      DOCUMENTED ("BY"):
             SOURCE
      EQUATED ("TO"):
             SYNONYM
      TRACED ("FROM"):
             DECISION
             ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
      ARTIFICIALITY:
             ARTIFICIAL
             VALIDATION
             IMPLEMENT_APPROXIMATELY
             IMPLEMENT_PRECISELY
      COMPLETENESS:
             INCOMPLETE
             COMPLETE
             CHANGEABLE
      DESCRIPTION:
             TEXT
      ENTERED_BY:
             TEXT
```

```
ELEMENT_TYPE:  MESSAGE
   LEGAL RELATIONSHIPS:
     IMPLEMENTS:
          VERSION
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     FORMED ("BY"):
          ALPHA
     MADE ("BY"):
          DATA
          FILE
     PASSED ("THROUGH"):
          INPUT_INTERFACE
          OUTPUT_INTERFACE
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     ARTIFICIALITY:
          ARTIFICIAL
          VALIDATION
          IMPLEMENT_APPROXIMATELY
          IMPLEMENT_PRECISELY
     COMPLETENESS:
          INCOMPLETE
          COMPLETE
          CHANGEABLE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
```

```
ELEMENT_TYPE:  ORIGINATING_REQUIREMENT
   LEGAL RELATIONSHIPS:
     IMPLEMENTS:
          VERSION
     INCORPORATES:
          ORIGINATING_REQUIREMENT
     TRACES ("TO"):
          ALPHA
          DATA
          DECISION
          ENTITY_CLASS
          ENTITY_TYPE
          EVENT
          FILE
          INPUT_INTERFACE
          MESSAGE
          OUTPUT_INTERFACE
          PERFORMANCE_REQUIREMENT
          R_NET
          SUBNET
          SUBSYSTEM
          UNSTRUCTURED_REQUIREMENT
          VALIDATION_PATH
          VALIDATION_POINT
          VERSION
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     INCORPORATED ("IN"):
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     COMPLETENESS:
          CHANGEABLE
          INCOMPLETE
          COMPLETE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
```

```
ELEMENT_TYPE:  OUTPUT_INTERFACE
   LEGAL RELATIONSHIPS:
     CONNECTS ("TO"):
          SUBSYSTEM
     IMPLEMENTS:
          VERSION
     PASSES:
          MESSAGE
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     ARTIFICIALITY:
          ARTIFICIAL
          VALIDATION
          IMPLEMENT_APPROXIMATELY
          IMPLEMENT_PRECISELY
     COMPLETENESS:
          INCOMPLETE
          COMPLETE
          CHANGEABLE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
```

```
ELEMENT_TYPE:  PERFORMANCE_REQUIREMENT
   LEGAL RELATIONSHIPS:
     CONSTRAINS:
          VALIDATION_PATH
     IMPLEMENTS:
          VERSION
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     COMPLETENESS:
          CHANGEABLE
          INCOMPLETE
          COMPLETE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
     TEST:
          TEXT
```

```
ELEMENT_TYPE:  R_NET
   LEGAL RELATIONSHIPS:
     IMPLEMENTS:
          VERSION
     DOCUMENTED ("BY"):
          SOURCE
     ENABLED ("BY"):
          EVENT
          INPUT_INTERFACE
     EQUATED ("TO"):
          SYNONYM
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     ARTIFICIALITY:
          ARTIFICIAL
          VALIDATION
          IMPLEMENT_APPROXIMATELY
          IMPLEMENT_PRECISELY
     COMPLETENESS:
          INCOMPLETE
          COMPLETE
          CHANGEABLE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
   LEGAL STRUCTURE ELEMENT_TYPES:
     ALPHA
     EVENT
     INPUT_INTERFACE
     OUTPUT_INTERFACE
     SUBNET
     VALIDATION_POINT
```

```
ELEMENT_TYPE:  SOURCE
   LEGAL RELATIONSHIPS:
     DOCUMENTS:
          ALPHA
          DATA
          DECISION
          ENTITY_CLASS
          ENTITY_TYPE
          EVENT
          FILE
          INPUT_INTERFACE
          MESSAGE
          ORIGINATING_REQUIREMENT
          OUTPUT_INTERFACE
          PERFORMANCE_REQUIREMENT
          R_NET
          SUBNET
          SUBSYSTEM
          UNSTRUCTURED_REQUIREMENT
          VALIDATION_PATH
          VALIDATION_POINT
          VERSION
     EQUATED ("TO"):
          SYNONYM
   LEGAL ATTRIBUTES:
     COMPLETENESS:
          CHANGEABLE
          INCOMPLETE
          COMPLETE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
```

```
ELEMENT_TYPE:  SUBNET
   LEGAL RELATIONSHIPS:
     IMPLEMENTS:
          VERSION
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     ARTIFICIALITY:
          ARTIFICIAL
          VALIDATION
          IMPLEMENT_APPROXIMATELY
          IMPLEMENT_PRECISELY
     COMPLETENESS:
          INCOMPLETE
          COMPLETE
          CHANGEABLE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
   LEGAL STRUCTURE ELEMENT_TYPES:
     ALPHA
     EVENT
     OUTPUT_INTERFACE
     SUBNET
     VALIDATION_POINT
```

```
ELEMENT_TYPE:  SUBSYSTEM
   LEGAL RELATIONSHIPS:
     IMPLEMENTS:
          VERSION
     CONNECTED ("TO"):
          INPUT_INTERFACE
          OUTPUT_INTERFACE
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     COMPLETENESS:
          CHANGEABLE
          INCOMPLETE
          COMPLETE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
```

```
ELEMENT_TYPE:  SYNONYM
   LEGAL RELATIONSHIPS:
     EQUATES ("TO"):
          ALPHA
          DATA
          DECISION
          ENTITY_CLASS
          ENTITY_TYPE
          EVENT
          FILE
          INPUT_INTERFACE
          MESSAGE
          ORIGINATING_REQUIREMENT
          OUTPUT_INTERFACE
          PERFORMANCE_REQUIREMENT
          R_NET
          SOURCE
          SUBNET
          SUBSYSTEM
          UNSTRUCTURED_REQUIREMENT
          VALIDATION_PATH
          VALIDATION_POINT
          VERSION
   NO LEGAL ATTRIBUTES
```

```
ELEMENT_TYPE:  UNSTRUCTURED_REQUIREMENT
   LEGAL RELATIONSHIPS:
     IMPLEMENTS:
          VERSION
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     COMPLETENESS:
          CHANGEABLE
          INCOMPLETE
          COMPLETE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
```

```
ELEMENT_TYPE:  VALIDATION_PATH
   LEGAL RELATIONSHIPS:
     IMPLEMENTS:
          VERSION
     CONSTRAINED ("BY"):
          PERFORMANCE_REQUIREMENT
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     ARTIFICIALITY:
          ARTIFICIAL
          VALIDATION
          IMPLEMENT_APPROXIMATELY
          IMPLEMENT_PRECISELY
     COMPLETENESS:
          INCOMPLETE
          COMPLETE
          CHANGEABLE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
     MAXIMUM_TIME:
          NUMERIC
     MINIMUM_TIME:
          NUMERIC
     UNITS:
          NAMED
   LEGAL PATH ELEMENT_TYPES:
     EVENT
     VALIDATION_POINT
```

```
ELEMENT_TYPE:  VALIDATION_POINT
   LEGAL RELATIONSHIPS:
     IMPLEMENTS:
          VERSION
     RECORDS:
          DATA
          FILE
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     ARTIFICIALITY:
          ARTIFICIAL
          VALIDATION
          IMPLEMENT_APPROXIMATELY
          IMPLEMENT_PRECISELY
     COMPLETENESS:
          INCOMPLETE
          COMPLETE
          CHANGEABLE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
```

```
ELEMENT_TYPE:  VERSION
   LEGAL RELATIONSHIPS:
     DOCUMENTED ("BY"):
          SOURCE
     EQUATED ("TO"):
          SYNONYM
     IMPLEMENTED ("BY"):
          ALPHA
          DATA
          DECISION
          ENTITY_CLASS
          ENTITY_TYPE
          EVENT
          FILE
          INPUT_INTERFACE
          MESSAGE
          ORIGINATING_REQUIREMENT
          OUTPUT_INTERFACE
          PERFORMANCE_REQUIREMENT
          R_NET
          SUBNET
          SUBSYSTEM
          UNSTRUCTURED_REQUIREMENT
          VALIDATION_PATH
          VALIDATION_POINT
     TRACED ("FROM"):
          DECISION
          ORIGINATING_REQUIREMENT
   LEGAL ATTRIBUTES:
     COMPLETENESS:
          CHANGEABLE
          INCOMPLETE
          COMPLETE
     DESCRIPTION:
          TEXT
     ENTERED_BY:
          TEXT
```

D.5 RSL TRANSLATION DIAGNOSTIC MESSAGES

Diagnostics from the RSL translation function are printed as an up-arrow (↑) pointing to the command symbol at which the error was first detected and an error number. The error may be detected one or more symbols beyond the actual error.

When an error occurs in the first sentence of a command (i.e., the sentence which identifies the subject element), the results are difficult to predict. The translator may not recognize the sentence as beginning a new command, will ignore the sentence and proceed to apply the succeeding sentences to the previous subject element as though no new command had begun.

Whenever a syntax error is detected (error numbers 12-199), all input text between the detection of the syntax error and the recovery from that error (error number 200) is ignored. No attempt is made to check the syntax of this text nor are any actions performed on the ASSM.

The diagnostics output during RSL translation are listed below. (This list also contains the diagnostics for the RSL Extension (RSLXTND) function.) The majority of the errors will cause the sentence to be ignored (i.e., no action will be taken in the ASSM). Those marked with (I) are informative only; the action specified in the command will be taken. Those marked with (F) will cause termination of translation and the return of control to the REVS Executive.

Error No.	Interpretation
0	The parse stack is not empty at the end of the translation. (This usually indicates a severe syntax error.) (F)
1	Parse stack overflow. Reduce the complexity of the statement. (F)
5	Illegal character or illegal two-character operator.
6	Integer too large (i.e., larger than 9 digits).
7	Too many significant figures in a real number (i.e., more than 9 digits).
8	Real number too large (i.e., absolute value greater than 1.0E74).
11	More than 50 errors in one line. Only the first 50 are listed.

Error No.	Interpretation
12-199	Syntax Error.
200	Recovery from previous syntax error. (I)
201	A comment was expected. (I)
202	Comment not allowed here. (I)
203	A period is missing at the end of the input file. (I)
204	Duplicate relationship instance in the ASSM. (I)
400	An instance of this attribute already exists.
401	The element type is not an applicable element type for this attribute.
402	The given element is associated with a node of a STRUCTURE or PATH.
403	The element type is an applicable element type for an attribute.
404	Illegal sentence in an attribute definition.
405	Attribute definition sentence misplaced.
406	OR node ordinal exceeds maximum of 9999.
407	Error in a conditional expression.
408	Complementary relation name is used in a relation definition header.
409	Element is not of type DATA.
410	Degenerate logical or arithmetic expression.
411	Sentence appears after a deletion command.
413	Duplicate attribute instance.
414	Duplicate definition for an attribute name.
415	Element type is already in the ASSM.
416	Duplicate element type list member.
417	Duplicate NET/PATH structure applicability.
418	Duplicate ordinals on two branches of the same OR node.

Error No.	Interpretation
419	Duplicate definition for a relation name.
421	Duplicate relation object on object list.
422	Element already has an associated STRUCTURE or PATH.
423	Duplicate attribute legal value. (I)
424	Illegal sentence in element definition.
425	Element definition sentence misplaced.
426	An element exists with this element type.
427	Node element is not of a type defined with structure applicability NET.
428	Illegal sentence in an element type definition.
429	An element of the given type is used on a NET or PATH.
430	Element type definition sentence misplaced.
431	"RECORD" must be preceded by FILE name.
432	Element name in the body of a FOR EACH node is not of type ALPHA or SUBNET.
433	Element is not of type FILE, ENTITY_TYPE, or ENTITY_CLASS.
434	Illegal syntax following DO.
435	An INPUT_INTERFACE begins a branch.
436	An INPUT_INTERFACE follows a node.
437	Illegal value for this attribute.
438	Illegal complementary relation name for this relation.
439	A declaration removal must be within a modification command.
440	Name being renamed is not an element name.
441	Illegal object element type for a relation.
442	Illegal subject element type for a relation.
443	Element type name is incorrect for the element.
444	Invalid attribute name.

Error No.	Interpretation
445	Invalid element name.
446	Invalid element type name.
447	Invalid relation name.
448	Too many new (undefined) names. (This error should be ignored between a syntax error (12-199) and the recovery from that error (200).)
449	Name already in use.
450	A node follows a terminator node or OUTPUT_INTERFACE.
451	New (undefined) name was lost. (This error should be ignored between a syntax error (12-199) and the recovery from that error (200).)
452	The ALL EXCEPT form may not be used for removals.
453	No instance of this attribute exists for the given element.
454	Given attribute legal value does not exist.
455	No object list in a relation declaration.
456	A node is not allowed in a structure removal.
457	Element type has no NET/PATH structure applicability.
458	Element does not have an associated PATH.
459	The given relationship instance does not exist.
460	Element does not have an associated STRUCTURE.
461	No type specified for undefined element.
462	No valid subject for the command.
463	No element type list is specified.
464	An attribute value is not specified.
465	Element is the object of a relation instance, therefore it cannot be deleted.
466	Ordinal and non-ordinal branches are mixed in an OR node.
467	Illegal sentence in relation definition.

Error No.	Interpretation
468	Relation definition sentence misplaced.
469	A RETURN may not appear on an R_NET.
470	An instance of this relationship exists.
471	The element type is a legal object element type of a relationship, therefore the element type definition cannot be deleted.
472	The element type is a legal subject element type of a relationship, therefore the element type definition cannot be deleted.
473	Structure has fewer than two nodes.
474	Structure has no terminator node or OUTPUT_INTERFACE.
475	Command subject is not of type R_NET or SUBNET so a structure declaration is illegal.
476	Element is the subject of a relation instance, and cannot be deleted.
477	The given SYNONYM already EQUATES to an element.
478	Terminating and non-terminating branches are mixed within an AND/OR node.
479	Name is undefined.
480	Node element is not of a type defined with structure applicability PATH.
481	Command subject is not of type VALIDATION_PATH, so PATH declaration is illegal.
482	An attribute value is specified where none is allowed. (I)
483	A node is not allowed in a PATH removal.
484	PATH has no nodes.
485	Zero ordinal.
486	Element is not of type ENTITY_TYPE or ENTITY_CLASS.
487	New type is redundant on a RETYPE.
488	An INPUT_INTERFACE is illegal on a SUBNET.
489	SUBNET has no RETURN.

Error No.	Interpretation
490	SUBNET has more than one RETURN.
491	RETYPE of an OUTPUT_INTERFACE which appears on a structure is illegal.
492	Element is not of type DATA or ENTITY_CLASS.
493	Element has a value for TYPE other than ENUMERATION.
494	Element is not of type ENTITY_TYPE.
495	Enumerated value name required.
496	Permission to use extensions not established.
497	Control permission required but not established.
498	A permission is already associated with the identifier.
499	No permission is associated with the identifier.
500	Translator was not invoked via RSLXTND.
501	Last control permission cannot be rescinded with outstanding extension permissions.
502	Extension permission cannot be established without any control permissions.
503	No non-empty branch on a CONSIDER OR node.
504	More than one empty branch on a CONSIDER OR node.
505	Duplicate name in consider conditional.
506	Comma, colon, or semicolon in conditional expression.
600	Name reserved by ASSM. (F)
601	Error in initialization. (F)
602	Error in REVS Executive request. (F)
603	Error in ASSM access routine. (F)
604	Required input file DONNEES is missing or empty. (F)
605	Required input file RSLDICT is missing, empty, or incomplete. (F)
606	Null ASSM, RSLXTND required for language definition. (F)

APPENDIX E

RNETGEN MESSAGES

A DANGLING STRUCTURE HAS BEEN DETECTED, SAVE REJECTED

This message results when, in an attempt to save a structure, an incomplete structure was detected, i.e., a node with no predecessor or successor. The structure is displayed with the node in question located at the center of the structure display area surrounded by a red, blue, and white square. The structure remains in the ASSM in temporary status.

A DO-NOTHING BRANCH DETECTED ON REJOINING AND, SAVE REJECTED

This message results when, in an attempt to save a structure, a null AND branch was detected, i.e., the successor node of an AND node is its matching rejoining AND node. The structure is displayed with the AND node in question centered in the structure display area surrounded by a red, blue, and white square. The structure remains in temporary status in the ASSM.

CALCOMP REQUEST COMPLETED, CONTINUE

This message is displayed upon completion of a CALCOMP menu operation. The message informs the user that the selected structure has been successfully plotted on CALCOMP and that further processing may now continue.

CANNOT EXECUTE RNETGEN FUNCTION IN OFFLINE MODE

This message will appear on REVS.OUT if the user attempts to execute the RNETGEN function while in the off-line mode. RNETGEN terminates and returns control to the REVS Executive.

COLOR HAD BEEN CHANGED ON SELECTED NODE, CONTINUE

This message is displayed merely to inform the user that the node color-change operation is complete and that further processing may now continue.

COMMENT HAS BEEN ATTACHED TO SELECTED NODE, CONTINUE

This message is displayed upon completion of a comment node menu operation. The message informs the user that the comment was successfully attached to the node in the ASSM and that further processing may now continue.

COMMENT MUST BEGIN WITH (*, INPUT REJECTED

This message results when an attempt to enter a comment which does not begin with the character string (* is made. The input is rejected, however the comment node menu selection remains in force.

COMMENT WAS REMOVED, IF INDEED, ONE EXISTED

This message results from the REMOVE comment operation. It is intended to inform the user that the selected operation is complete and that other processing may now continue.

CONSIDER - DATA, ENTITY_CLASS, NEITHER - SELECT VIA TRACKBALL

This message is displayed after having selected the point on the screen to locate an OR node. The user must respond by selecting one of the entries in the message using the trackball.

DISPLAY BRANCH IS COMPLETE, CONTINUE

This message is displayed merely to inform the user that the operation is complete and that further processing may now continue.

DO YOU WANT STANDARD DOCUMENT SIZE? KEYIN YES OR NO

This message is displayed when the user selects the CALCOMP menu entry. The keyboard is enabled and the user must respond with a YES (Y) or NO (N) keyboard entry. If the response is YES, the CALCOMP display will be 8-1/2 x 11 inch document size, otherwise the user will be required to supply the desired document size.

DOES BRANCH HAVE CONDITIONAL EXPRESSION? KEYIN YES OR NO

This message is displayed when an attempt is made to connect a FOR EACH node with its successor node. The keyboard is enabled and the user must respond with a YES (Y) or NO (N) keyboard entry. If the response is YES, the user will be required to enter the conditional expression via the keyboard.

DOES BRANCH HAVE ORDINAL VALUE? KEYIN YES OR NO

This message is displayed when an attempt is made to create the initial branch of an OR node. The keyboard is enabled and the user is required to respond with a YES (Y) or NO (N) keyboard entry. If the response is YES, the user will be required to enter the ordinal value.

DUPLICATE ORDINAL, INPUT REJECTED, ENTER TB TO CONTINUE

This message results if there already exists a branch from the current OR node which has an ordinal value equal to the one which was just input. The ordinal input is rejected and the user is required to re-input a different ordinal value.

ELEMENT ALREADY IN ASSM. IS IT THE ONE? KEYIN YES OR NO

This message results when the element name keyed-in by the user is already in the ASSM and its type agrees with that reflected by the menu selection. The keyboard is enabled and the user is required to respond with YES or NO. If the response is YES and the menu selection was a structure type, the structure, if one exists, is retrieved from the ASSM and displayed in the structure display area. If the response is YES and the menu selection was a node type, the node is displayed at the selected position in the structure display area. The first three characters of the associated element name is also displayed on the node. If the response is NO, the element name which was keyed-in is ignored and the current menu selection remains in force.

ELEMENT HAS BEEN ENTERED INTO ASSM

If the user responded with a YES to the previous message (i.e., ELEMENT NOT IN ASSM. DO YOU WANT IT ENTERED? KEYIN YES OR NO), the element is entered into the ASSM and the above message is displayed.

ELEMENT HAS NOT BEEN ENTERED INTO ASSM

If the user responded with a NO to the previous message (i.e., ELEMENT NOT IN ASSM. DO YOU WANT IT ENTERED? KEYIN YES OR NO), then the element is not entered into the ASSM and the input is ignored; however, the current menu selection remains in force.

ELEMENT HAS TYPE OTHER THAN ENUMERATION, INPUT REJECTED

If element type DATA was selected for the CONSIDER OR node and the supplied element name has the attribute TYPE with a value other than ENUMERATION, this message is displayed. The element name is ignored and the node screen position must be reselected by the user.

ELEMENT IN EXPRESSION IS OF INCORRECT TYPE, INPUT REJECTED

If the conditional expression keyed-in for an OR/FOR EACH branch contains ASSM elements of element type other than DATA, then this message results. The desired branch connection is ignored and the user must again identify the nodes to be connected and re-input the conditional expression.

ELEMENT IS IN ASSM AND IS OF INCOMPATIBLE TYPE, INPUT REJECTED

This message results when the element name keyed-in by the user is already in the ASSM and its type does not agree with that reflected by the menu selection. The input is rejected and the current menu selection remains in force.

ELEMENT NOT IN ASSM. DO YOU WANT IT ENTERED? KEYIN YES OR NO

This message results if the element name keyed-in does not currently exist in the ASSM. The user must respond via the keyboard with either YES (Y) or NO (N).

ELEMENT_TYPE DOES NOT EXIST IN THE ASSM, INPUT REJECTED

This message results if an element type keyed-in by the user cannot be found in the ASSM. The menu selection is ignored.

ELEMENT_TYPE UNDEFINED FOR THIS NODE TYPE, INPUT REJECTED

This message results when, in an attempt to enter a node for a selected node type, it is determined that there exists no definition in the ASSM for such an element. For instance, if an attempt to enter an alpha node is made but there exists no element type ALPHA in the ASSM the above message is displayed and the request is ignored.

ENTER COMMENT SEGMENT

This message is displayed when in the comment node menu operation and a node has been selected to comment. The keyboard is enabled and the user must keyin a comment line. The entire comment must be enclosed with (* and *).

ENTER CONDITIONAL EXPRESSION SEGMENT

This message is displayed when attempting to connect an OR/FOR EACH node with its successor. The keyboard is enabled and the user must keyin the desired conditional expression. The entire expression must be enclosed within parentheses, except for an OTHERWISE expression.

ENTER DESIRED ELEMENT_TYPE VIA THE KEYBOARD

If the user has selected a node type of OTHER in the menu, the above message is displayed. The user responds with a keyboard entry of the appropriate element type.

ENTER, REMOVE, OR DISPLAY COMMENT ON NODE, SELECT VIA TB

This message results from the selection of a node after having selected the comment node menu operation. The user must respond with a trackball selection of the desired operation within the displayed message.

ENTIRE COMMENT ON SELECTED NODE HAS BEEN DISPLAYED

This message is displayed merely to inform the user that the display comment operation is complete for the selected node.

ERROR IN INPUT, TRY AGAIN

This message results when a non-numeric value is input as a CALCOMP document size. The input is ignored and the keyboard is re-enabled for input.

FOR EACH - FILE, ENTITY_CLASS, ENTITY_TYPE - SELECT VIA TB

This message is displayed when a FOR EACH node is entered on a structure. The trackball is enabled and the user must respond with a trackball selection of one of the three entries in the message in order to designate the element type for the element associated with the FOR EACH node.

ILLEGAL MENU SELECTION

This message will result if an attempt is made to zoom-in on a structure which is not in the zoomed-out mode, i.e., the structure has not first been zoomed-out. The message will also occur if an attempt is made to process a structure which is non-existent. Selection is ignored.

ILLEGAL NODE FOR CURRENT STRUCTURE

This message is displayed if an attempt is made to enter a node which is not allowed for the current structure type (e.g., a return node on an R-Net structure is illegal). Another menu selection should be made at this point.

ILLEGAL NODE SELECTION, RESTART SELECTION SEQUENCE

This message may result for several different reasons as follows:

(a) an attempt to delete a node which has successors which have no graphics coordinate data.

(b) an attempt to disconnect nodes which are not actually connected.

(c) an attempt to connect nodes which cannot be legally connected.

The selection sequence is ignored, however the current menu selection remains in force.

ILLEGAL ORDINAL VALUE, INPUT REJECTED, ENTER TB TO CONTINUE

This message results if an attempt is made to input an ordinal value whose characters are not numeric. The input is rejected and the user is required to re-input the ordinal value correctly via the keyboard.

ILLEGAL REJOINING AND/OR NODE, SAVE REJECTED

This message results when, in an attempt to save a structure, either an unmatched rejoining AND/OR node was detected or an attempt to mix splitting and rejoining branches from the same OR node was detected. In either case, the structure is displayed with the node in question centered in the structure display area surrounded by a red, blue, and white square. The structure remains in the ASSM in temporary status.

ILLEGAL SUCCESSOR NODE, RESTART SELECTION SEQUENCE

This message will result if, when attempting to connect two nodes, it has been determined that the desired successor node cannot legally follow the selected predecessor node. The desired connection is ignored, however, the current menu selection remains in force (i.e., the user may reselect two nodes).

ILLEGAL SYNTAX, INPUT REJECTED

This message results if the ASSM element name typed in via the keyboard contains illegal characters or does not begin with an alphabetic character. The message also results if the syntax for the CONSIDER OR node conditional expression is in error.

ILLEGAL TRACKBALL SELECTION

This message will result if the user selects a point on the screen outside the structure display area when indeed the selected point should be within the structure display area. The input is ignored, however the current menu selection remains in force.

IS PREDECESSOR NODE A REJOINING -OR- NODE, KEYIN YES OR NO

This message is displayed when the predecessor node of two nodes to be connected is an OR node and no implicit determination can be made as to its type -- rejoining or non-rejoining OR node. The keyboard is enabled and the user must respond with a YES (Y) or NO (N) keyboard entry.

IS THIS A REJOINING -OR/AND- NODE?

This message is displayed when the user attempts the comment node menu operation on an OR/AND node. The user must respond by typing in YES (Y) or NO (N) via the keyboard. If the response is YES (Y) the operation will be ignored.

KEYIN DOCUMENT HEIGHT (INCHES)

This message follows a request for entry of the document width when the user chooses not to use standard document size for CALCOMP output. The keyboard is enabled and the user inputs a one to four digit number (real or integer) to be used for the height of the CALCOMP output. Any number larger than 29.0 will be reduced to 29.0.

KEYIN DOCUMENT WIDTH (INCHES)

The message results if the user chooses not to use standard document size for CALCOMP output. The keyboard is enabled and the user inputs a one to four digit number (integer or real) to be used for the width of the CALCOMP output. Any number larger than 50.0 will be set to 50.0.

KEYIN ELEMENT NAME OF DESIRED ELEMENT

This message is displayed when the user selects a structure type or when the user selects a screen position in the structure display area after having selected the desired node type. The keyboard is enabled and the user is required to enter the element name of the desired element.

KEYIN ENTITY_TYPE ELEMENT(S) ASSOCIATED WITH THIS BRANCH ( )

This message is displayed when the user attempts to connect a CONSIDER OR node with its successor. The CONSIDER OR node is associated with an ASSM element of type ENTITY_CLASS. The user responds by typing in the ENTITY_TYPE element(s) associated with the selected branch.

KEYIN ORDINAL VALUE

This message is displayed when, in creating a branch of an OR node, it has been determined that an ordinal value is required. The keyboard is enabled and the user must respond with a one to four digit keyboard entry.

KEYIN RANGE VALUE(S) ASSOCIATED WITH THIS BRANCH ( )

This message is displayed when the user attempts to connect a CONSIDER OR node with its successor. The CONSIDER OR node is associated with an ASSM element of type DATA. The user responds by typing in the RANGE values associated with the selected branch.

LOOP DETECTED IN THE STRUCTURE, SAVE REJECTED

This message results when, in an attempt to save a structure, a loop was detected by the structure analyzer. The structure is displayed with the node in question centered in the structure display area surrounded by a red, blue, and white square. The structure remains in the ASSM in temporary status.

MOVE HAS BEEN COMPLETED, CONTINUE

This message is displayed upon completion of a move node menu operation. The message is intended to inform the user that the node was successfully moved to its new screen position, and that further processing may now continue.

MOVE SUBTREE OPERATION IS COMPLETE, CONTINUE

This message is displayed upon completion of the move subtree menu operation. The message informs the user that the move subtree operation was successfully completed and that further processing may now continue.

NAME ALREADY IN ASSM AS SOME OTHER RSL NAME, NODE REJECTED

This message results when an element name keyed in by the user already exists in the ASSM for some other use such as relationship name, attribute name, etc. The input is ignored; however, the current menu selection remains in force.

NO COMMENT EXISTS ON SELECTED NODE, REQUEST IGNORED

If the user has selected a node which contains no comments for display, the above message is displayed.

NO CONDITIONAL EXPRESSION ON THIS BRANCH

This message results if, after having selected the display branch menu operation, the selected branch contains no conditional expression. The input is then rejected.

NO CURRENT STRUCTURE ASSOCIATED WITH ELEMENT

This message results when an attempt is made to display the structure for an element selected by the user and no structure exists in the ASSM for that element. An entry node is created for the structure and displayed at the upper center portion of the structure display area. The user may then add to the structure by selecting desired node types via the menu.

NO GRAPHICS DATA ON STRUCTURE

This message is displayed when the structure for the element name supplied by the user has no graphics coordinate data for its associated nodes. The structure was created and entered in the ASSM via the RSL translator.

NO ORDINAL, ENTER TRACKBALL FOR CONDITIONAL EXPRESSION

If the user has selected the display branch menu operation and the selected branch does not contain an ordinal, then the above message results.

NO STRUCTURE ASSOCIATED WITH SELECTED ELEMENT, INPUT REJECTED

This message results if an attempt is made to generate CALCOMP output for a non-existent structure. The input request is ignored.

NO STRUCTURE AVAILABLE, SAVE REJECTED

This message results when an attempt is made to save a nonexistent structure. The request is simply ignored.

NODE DEFINITION IS INCOMPLETE, SAVE REJECTED

This message results when, in an attempt to save a structure, an AND/OR node was detected as having only one incoming and one outgoing branch. The structure is displayed with the node in question centered in the structure display area surrounded by a red, blue, and white square. The structure remains in the ASSM in temporary status.

NODE OVERLAP IN STRUCTURE, REPEAT SELECTION SEQUENCE

This message results when an attempt is made to create/locate a node which would overlap an already existing node in the structure display area. The user selection of the node location is ignored; however, the current menu selection remains in force.

NODE WILL NOT FIT ON DRAWING AREA, TRY AGAIN

This message results when an attempt is made to create/locate a node either outside the structure display area or too near the border surrounding the structure display area. The user entry of the node location is ignored but the current menu selection remains in force.

NODES HAVE BEEN DISCONNECTED, CONTINUE

This message is displayed upon completion of a disconnect nodes menu operation. The message informs the user that the nodes were successfully disconnected in the ASSM and that further processing may now continue.

NON-EXISTENT NODE SELECTED, RESTART SELECTION SEQUENCE

This message results when, in an attempt to select a node in the structure display area, the screen position selected actually contains no node. The position selection(s) is ignored, however the current menu selection remains in force.

NOT EXACTLY ONE OTHERWISE BRANCH ON OR BRANCH, SAVE REJECTED

This message results when, in an attempt to save a structure, it was determined that an OR node (not a CONSIDER OR node) exists in the structure which does not have one and only one OTHERWISE branch. The structure is displayed with the OR node in question centered in the structure display area surrounded by a red, blue, and white square. The structure remains in the ASSM in temporary status.

NOT EXACTLY ONE RETURN NODE ON THIS SUBNET, SAVE REJECTED

If, in the process of attempting to save a SUBNET structure, the structure does not contain one and only one SUBNET return node, then the above message results. The user must correct the situation before the structure can be saved.

SCROLL IS COMPLETE, CONTINUE

This message is displayed upon completion of a SCROLL net menu operation. The message informs the user that the structure was successfully scrolled and that further processing may now continue.

SELECT FROM MENU WITH TRACKBALL

This message is displayed immediately upon entering the RNETGEN function. The only menu selection allowed at this point is one of the following: one of the structure types, color selection, or the STOP command.

SELECT NODE TO BE DISCONNECTED

This message is displayed after selecting the first of the two nodes to be disconnected. The user must respond with a trackball selection of the desired node to be disconnected from the previously selected node.

SELECT SUCCESSOR NODE

This message is displayed after selecting the first (predecessor) node of the two nodes which are to be connected. The user must respond with a trackball selection of the desired successor node.

SELECTED NODE HAS BEEN DELETED, CONTINUE

This message is displayed upon completion of a delete node menu operation. The message informs the user that the node was successfully deleted from the ASSM, and that further processing may now continue.

SELECTED NODE HAS NO ASSOCIATED ELEMENT, CONTINUE

This message results when in the display node menu operation and the selected node has no element associated with it in the ASSM. The selection is ignored and the current menu selection remains in force.

SELECT NODES HAVE BEEN CONNECTED, CONTINUE

This message is displayed upon completion of a connect nodes menu operation. The message informs the user that the nodes were successfully connected, and that further processing may now continue.

SELECT - ENTITY_CLASS, ENTITY_TYPE - ENTER VIA TRACKBALL

When positioning a SELECT node on the screen, the above message is displayed. The user must respond with a trackball selection of one of the two entries in the message.

SELECT - PROMPTER OR AUTOPLOT - VIA TRACKBALL

This message is displayed when the structure selected by the user has no graphics coordinate data for its associated nodes. The structure was created and entered in the ASSM via the RSL translator function. The user responds by selecting either the word PROMPTER or the word AUTOPLOT in the displayed message using the trackball. If the PROMPTER selection is made, the user will be requested to develop the structure graphics using the successor node menu operation. If AUTOPLOT is chosen, the graphics coordinate data are generated by REVS.

SELECT - TO POSITION - VIA TRACKBALL

This message results when attempting to move a node or to scroll the entire structure. The user responds by selecting, via the trackball, a position on the screen to which he wishes a previously selected node to be moved or, in the case of a scroll, to which he wishes a previously selected point on the structure to be moved (SCROLLED).

STRUCTURE DOES NOT HAVE AN ENTRY NODE, SAVE REJECTED

This message results when, in an attempt to save a structure, it has been determined that no entry node exists for the structure. The structure remains in the ASSM in temporary status.

STRUCTURE ALREADY HAS AN ENTRY NODE, SELECTION REJECTED

This message results if an attempt is made to enter more than one entry node for a structure. The input is ignored; however, the menu selection remains in force.

STRUCTURE HAS BEEN SAVED, CONTINUE

This message is displayed upon completion of a save net menu operation. The message is intended to inform the user that the structure was successfully entered in the ASSM in a permanent status, and that further processing may now continue.

STRUCTURE NEEDS ADDITIONAL GRAPHICS DATA, SAVE REJECTED

This message results when, in an attempt to save a structure, at least one node was detected to have no graphics coordinate data. The structure is displayed with the predecessor to the node in question centered in the structure display area surrounded by a red, blue, and white square. The structure remains in the ASSM in temporary status.

SUCCESSOR NODE HAS BEEN DISPLAYED, CONTINUE

This message is displayed upon completion of a successor node menu operation. The message is intended to inform the user that the successor node operation was successful, and that further processing may now continue.

SYNTAX ERROR WAS DETECTED IN EXPRESSION, INPUT REJECTED

This message results from a syntax error in a conditional expression on an OR/FOR branch. The line containing the error is displayed and the character in error is displayed in red. The desired branch connection and associated conditional expression is rejected. The user must re-indicate nodes to be connected and subsequently re-input the conditional expression correctly.

THERE ARE NO MORE SUCCESSOR NODES ON SELECTED NODE, CONTINUE

This message appears in the successor node menu operation when the selected node has no other successor nodes without graphics coordinate data. The node selection is ignored; however, the current menu selection remains in force.

UNDEFINED ELEMENT_TYPE FOR THIS STRUCTURE, INPUT REJECTED

This message results when the structure type selected by the user has no corresponding element type defined in the ASSM. For instance, if a subnet structure was selected from the menu and no element type SUBNET exists in the ASSM, then the above message is displayed and the request is ignored.

WARNING ... PREVIOUS STRUCTURE WAS NOT SAVED, RE-SELECT MENU

This message results from an attempt to begin/select another structure or to stop (exit RNETGEN) prior to saving the currently displayed structure. At this point the user may select any appropriate menu operation including the save net operation.

ZOOM-IN IS COMPLETE, CONTINUE

This message is displayed upon completion of a ZOOM-IN on net menu operation. The message is intended to inform the user that the ZOOM-IN operation was successful, and that further processing may now continue.

ZOOM-OUT IS COMPLETE, CONTINUE

This message is displayed upon completion of a ZOOM-OUT on net menu operation. The message informs the user that the ZOOM-OUT operation was successful and that further processing may now continue.

ZOOM-OUT OR ZOOM-IN ON STRUCTURE, SELECT VIA TRACKBALL

This message is displayed in order to provide the user with the option of displaying the selected structure in either its zoomed-out or its zoomed-in form. The user responds with an appropriate trackball selection.

APPENDIX F

RADX SUMMARY

F.1 RADX RCL SYNTAX

Table F.1 contains the syntax of the RADX command statements presented in the BNF notation described in Appendix A. Each entry in the table also includes the number of the section in this document which describes the corresponding syntax.

Figure F-1 shows the syntax in diagrammatic form.

Table F.1 RADX RCL Index

RADX RCL SYNTAX	SECTION NUMBER
<RADX command>::= <set definition> | <hierarchy definition> | <append definition> | <list set> | <list RSL> | <list permission> | <plot set> | <analyze set>	6.0
<set definition>::= SET set-name = <set description>	6.1
<set description>::= <set list> | <set combination> | <attribute qualification> | <relationship qualification> | <structure qualification> | <hierarchy qualification>	6.1
<set list>::= $\left\{\text{<set id>}\right\}_1^n$.	6.1.1
<set id>::= [SET] $\left\{\begin{array}{l}\text{ALL}\\ \text{ANY}\\ \text{set-name}\\ \text{element-name}\\ \text{element-type-name}\end{array}\right\}_1^1$	6.1
<set combination>::= <set id> $\left\{\begin{array}{l}\text{AND}\\ \text{OR}\\ \text{MINUS}\end{array}\right\}_1^1$ <set id>.	6.1.2
<attribute qualification>::= <set id> <connector> attribute-name $\left[[\text{<rel op>}]\left\{\begin{array}{l}\text{value-name}\\ \text{number}\\ \text{text-string}\end{array}\right\}_1^1\right]$.	6.1.3
<connector>::= <positive connector> | <negative connector>	6.1.3
<positive connector>::= WITH | WHERE | WHICH [IS] | IN | FROM | SUCH [THAT] | THAT [IS] | BY	6.1.3
<negative connector>::= WITHOUT | <positive connector> NOT | <positive connector> NO	6.1.3
<rel op>::= > | >= | = | <> | <= | <	6.1.3
<relationship qualification>::= <set id> <connector>[MULTIPLE] relation-name$\left\{\text{relation-optional-word}\right\}_0^n$[<set id>].	6.1.4
<structure qualification>::= <set id> <connector>[MULTIPLE] $\left\{\begin{array}{l}\text{REFERS}\\ \text{REFERRED}\end{array}\right\}_1^1$ $\left\{\text{relation-optional-word}\right\}_0^n$ [<set id>].	6.1.5
<hierarchy qualification>::= <set id> <connector> $\left\{\begin{array}{l}\text{HIER}\\ \text{HIERARCHY}\end{array}\right\}_1^1$ hierarchy-name.	6.1.7
<hierarchy definition>::= $\left\{\begin{array}{l}\text{HIER}\\ \text{HIERARCHY}\end{array}\right\}_1^1$ hierarchy-name = $\left\{\text{<set id> <binding relation> <set id>}\right\}_1^n$.	6.1.6
<binding relation>::= relation-name $\left\{\text{relation-optional-word}\right\}_0^n$ | REFERS $\left\{\text{relation-optional word}\right\}_0^n$ | REFERRED $\left\{\text{relation-optional word}\right\}_0^n$	6.1.6

Table F.1 RADX RCL Index (Continued)

RADX RCL SYNTAX	SECTION NUMBER
<append definition>::= APPEND $\left\{\begin{matrix}\text{ALL}\\ \text{ANY}\\ \text{element-type-name}\end{matrix}\right\}_1^1 \left\{\text{<append item>}\right\}_1^n$.	6.2.2
<append item>::= attribute-name | relation-name [relation-optional-word] | REFERS [relation-optional-word] | REFERRED [relation-optional-word] | ALL | NONE | STRUCTURE | ATTRIBUTE | RELATION | RELATIONSHIP | PRIMARY | COMPLEMENTARY	6.2.2
<list set>::= $\left\{\begin{matrix}\text{LIST}\\ \text{PUNCH}\end{matrix}\right\}_1^1$ <set id> [<hierarchy list option>].	6.2.1 6.2.3 6.2.7
<hierarchy list option>::= <positive connector> $\left\{\begin{matrix}\text{HIER}\\ \text{HIERARCHY}\end{matrix}\right\}_1^1$ hierarchy-name $\left[\text{<positive connector>} \left\{\begin{matrix}\text{MAP}\\ \text{GROUP}\\ \text{SEQUENCE}\end{matrix}\right\}_1^1\right]$	6.2.3
<list RSL>::= $\left\{\begin{matrix}\text{LIST}\\ \text{PUNCH}\end{matrix}\right\}_1^1$ RSL [<RSL item>].	6.2.5 6.2.7
<RSL item>::= element-type-name [SUMMARY] | relation-name | attribute-name | ALL | ELEMENT_TYPE | RELATIONSHIP | RELATION | ATTRIBUTE | SUMMARY	6.2.5
<list permission>::= $\left\{\begin{matrix}\text{LIST}\\ \text{PUNCH}\end{matrix}\right\}_1^1$ PERMISSION GIVEN control-permission-name.	6.2.6
<plot set>::= PLOT <set id> $\left\{\text{<plot size>}\right\}_0^n$.	6.2.4
<plot size>::= WIDTH [=] number | HEIGHT [=] number	6.2.4
<analyze set>::= ANALYZE [DATA_FLOW] <set id> $\left[\text{USING} \left\{\begin{matrix}\text{BETA}\\ \text{GAMMA}\\ \text{IMPLIED}\end{matrix}\right\}_1^1\right]$.	6.3

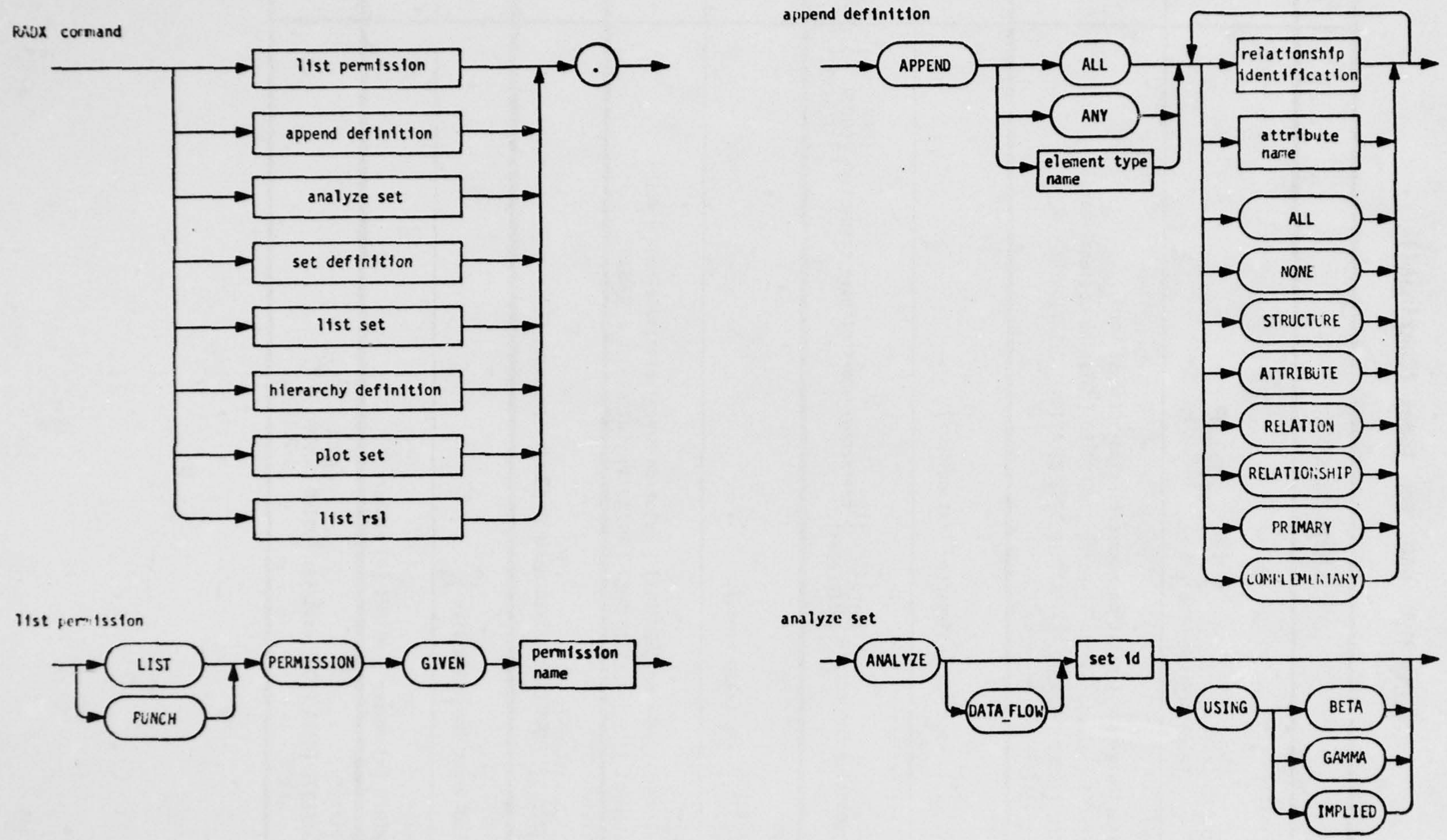
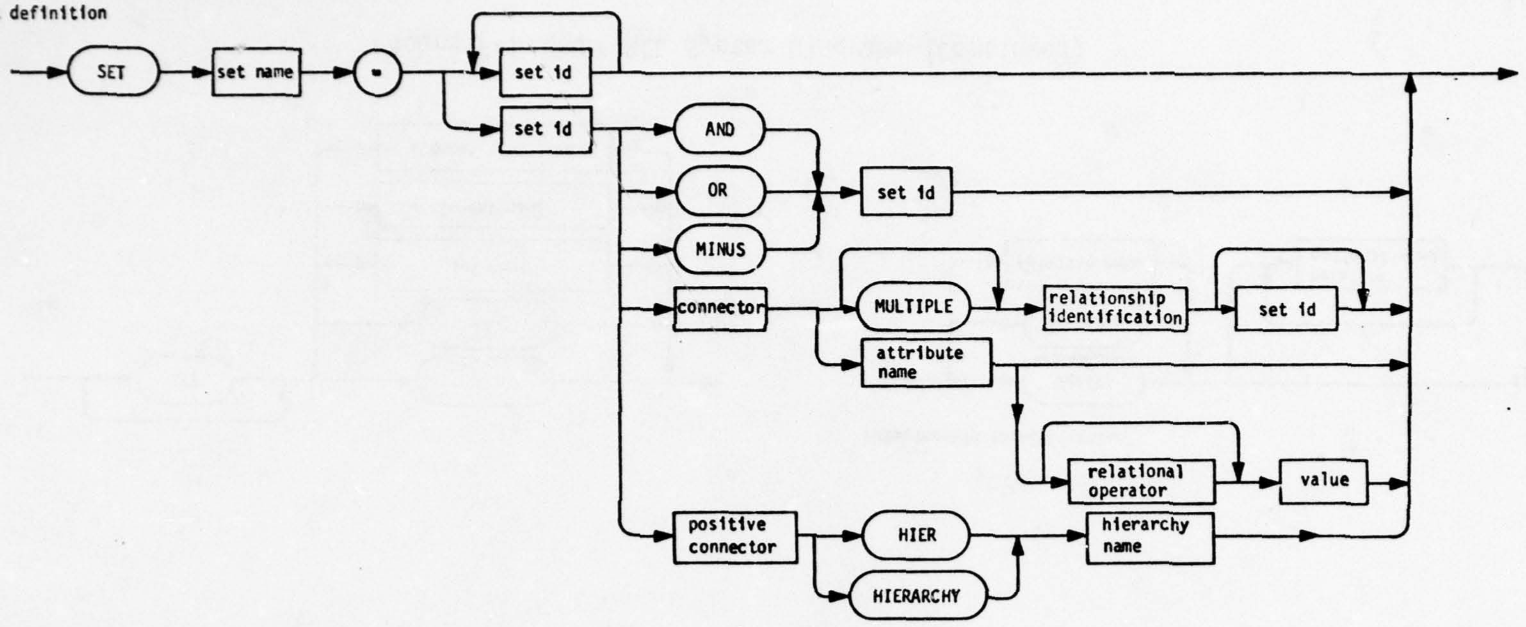


Figure F-1 RADX RCL Syntax Diagrams

set definition



list set

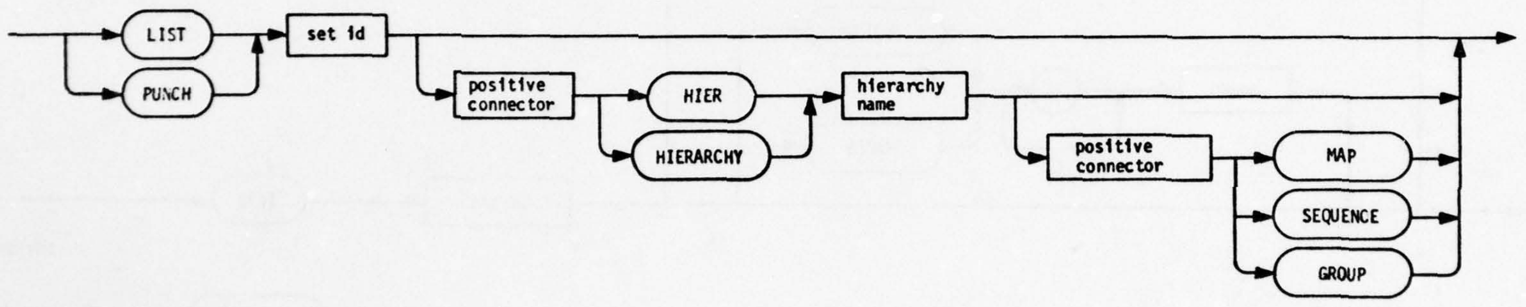


Figure F-1 RADX RCL Syntax Diagrams (Continued)

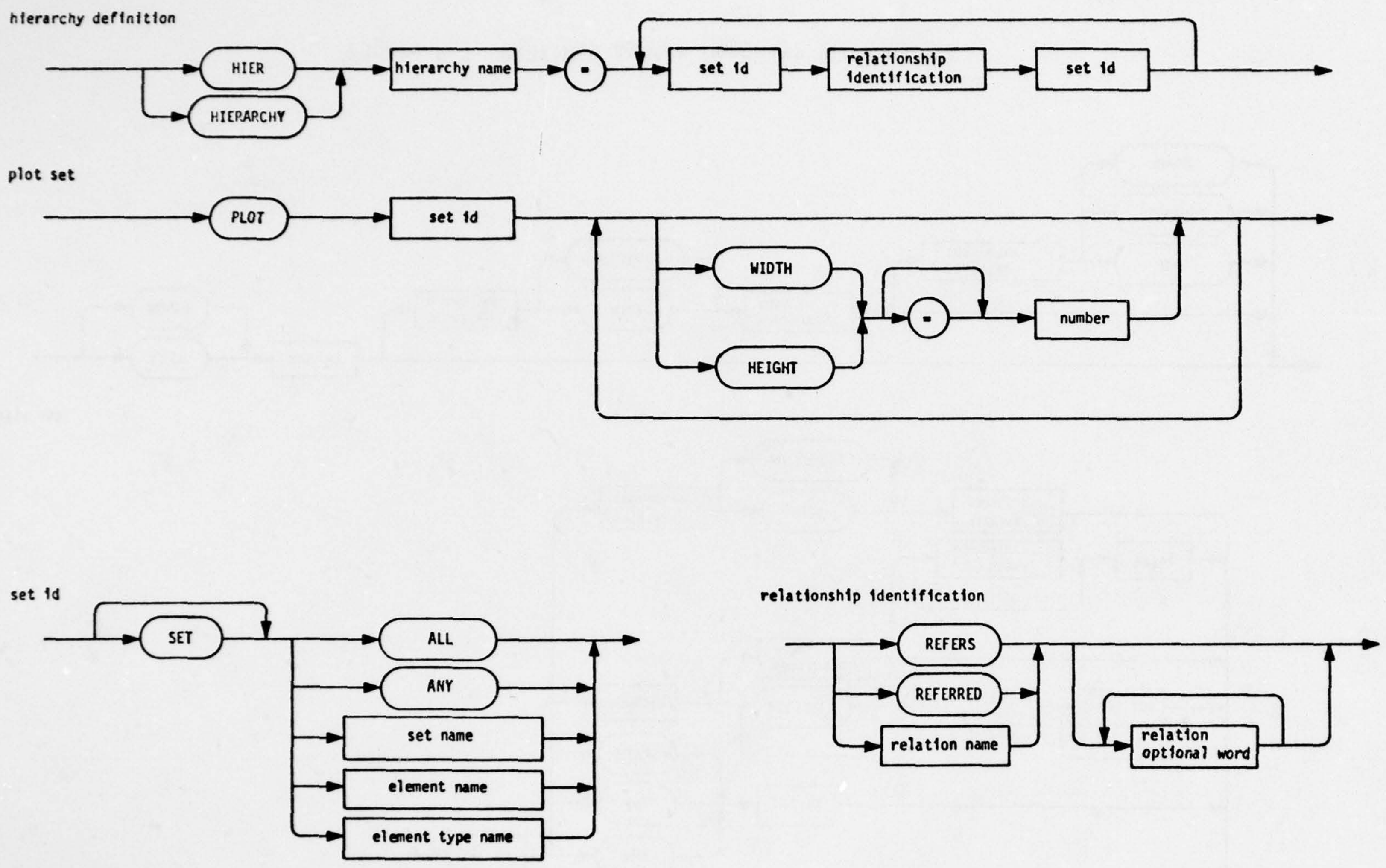
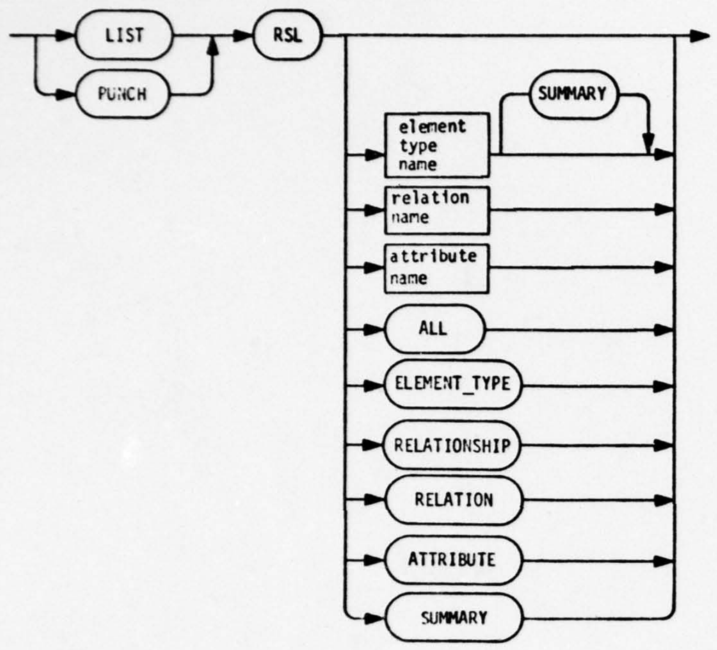
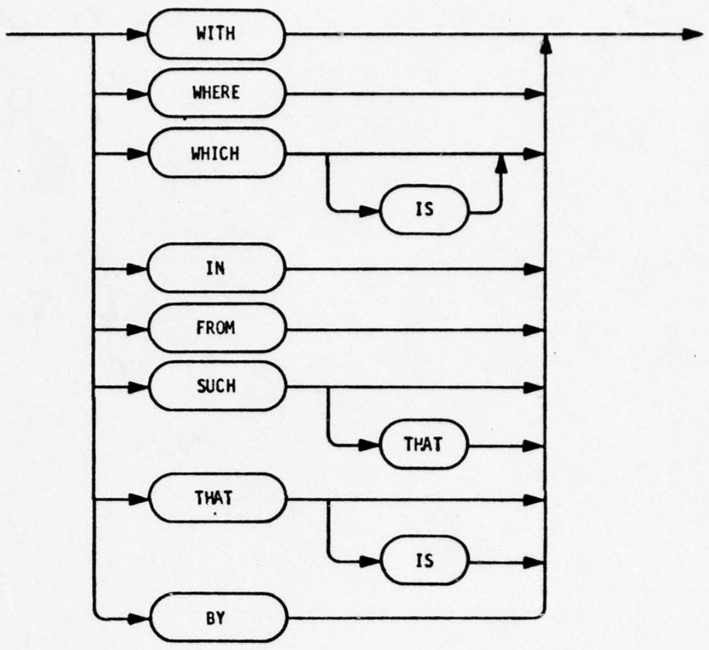


Figure F-1 RADX RCL Syntax Diagrams (Continued)

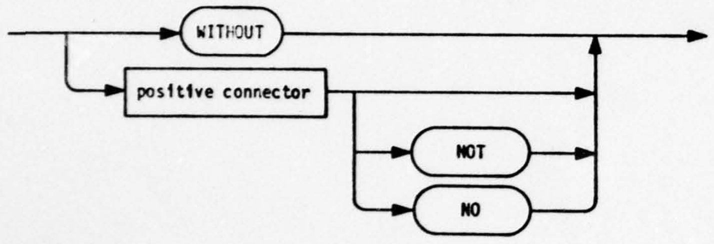
list rsl



positive connector



connector



relational operator

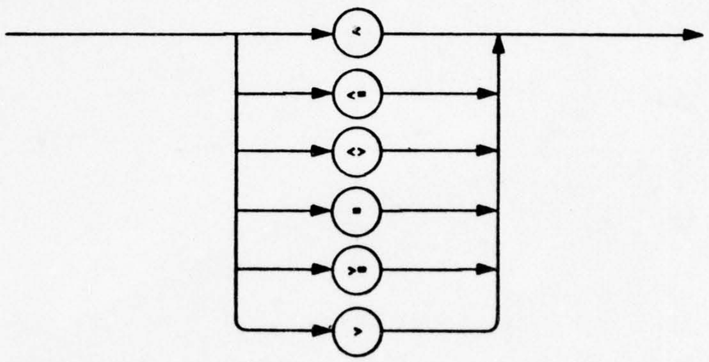


Figure F-1 RADX RCL Syntax Diagrams (Continued)

F.2 RADX DIAGNOSTIC MESSAGES

A summary of the error messages that can be issued by the RADX function is listed below. All error messages have the following standard format:

*ERROR XXXX description.

The XXXX is a unique four digit number that appears with the description of each of the following messages.

0060 ILLEGAL KEYWORD WAS SPECIFIED.

0061 BAD ANALYZE COMMAND.

0090 SET TO BE LISTED DOES NOT EXIST.

0091 BAD LIST SET COMMAND.

0092 NOT OPTION CANNOT BE APPLIED TO LIST BY HIERARCHY.

0093 INCOMPLETE LIST SET BY HIERARCHY COMMAND.

0094 REFERENCE TO UNDEFINED HIERARCHY.

0095 BAD LIST RSL SELECTION.

0096 INCORRECT LIST PERMISSION COMMAND.

0100 SET NAME CANNOT APPEAR IN THE ASSM.

0101 SET NAME MUST BE ALPHANUMERIC BEGINNING WITH ALPHA.

0102 SET NAME MUST NOT BE AN RCL RESERVED WORD.

0120 REFERENCE TO UNDEFINED SET IN SET DEFINITION STATEMENT.

0180 INCOMPLETE SET QUALIFICATION COMMAND.

0181 NOT OPTION ILLEGAL WHEN QUALIFYING BY HIERARCHY.

0182 REFERENCE TO UNDEFINED HIERARCHY.

0183 BAD QUALIFY SET COMMAND.

0184 REFERENCE TO UNDEFINED OBJECT SET IN SET QUALIFICATION.

0210 REFERENCE TO UNDEFINED SET.

0240 INCOMPLETE APPEND STATEMENT.

PRECEDING PAGE NOT FILMED BLANK

0241 REFERENCE TO UNDEFINED RSL ELEMENT TYPE.

0242 ILLEGAL SELECTION IN APPEND STATEMENT.

0280 SYMBOL STRING IS TOO LONG.

0281 CHARACTER STRING IS TOO LONG

0320 INPUT LINE BUFFER OVERFLOW (STATEMENT IS TOO LONG).

1080 AND-NODE USED TO SPLIT AND JOIN, OR NEITHER.

Can only be caused by system error.

1120 OR-NODE USED TO SPLIT AND JOIN, OR NEITHER.

Can only be caused by system error.

1200 AND-NODE USED TO SPLIT AND JOIN, OR NEITHER.

Can only be caused by system error.

1240 NO SUCCESSOR ON SPLIT OR-NODE.

Can only be caused by system error.

1241 OR-NODE USED BOTH AS SPLIT AND JOIN, OR NEITHER.

Can only be caused by system error.

1520 RADX ABORT FROM QQERRPRC.

Message issued when RADX has to abort because of system error.

1540 OUT OF SET STORAGE SPACE.

The user has exceeded the number of sets that can be defined in a single activation of RADX.

1660 OUT OF GLOBAL WORKING SET SPACE.

Can only be caused by RADX software error.

1790 ASSM DATA BASE ERROR DETECTED.

Indicates that RADX is unable to retrieve information from the ASSM.

1800 ATTRIBUTE VALUE IS ILL-FORMED.

1801 ATTRIBUTE RELATIONAL OPERATOR IS ILLEGAL FOR ATTRIBUTE VALUE.

1802 ATTRIBUTE VALUE EXPECTED.

1804 ATTRIBUTE VALUE SPECIFIED IS ILLEGAL.

1820 PRIMARY RELATION NAME STRING IS NOT IN THE ASSM.
Can only be caused by system error.

1821 PRIMARY RELATION OPTIONAL WORD STRING IS NOT IN THE ASSM.
Can only be caused by system error.

1825 PRIMARY RELATION INSTANCE IS NOT IN THE ASSM.
Can only be caused by system error.

1840 COMPLEMENTARY RELATION NAME STRING IS NOT IN THE ASSM.
Can only be caused by system error.

1841 COMPLEMENTARY RELATION OPTIONAL WORD STRING IS NOT IN ASSM.
Can only be caused by system error.

1842 PRIMARY RELATION FOR COMPLEMENTARY RELATION IS NOT IN ASSM.
Can occur because of an incomplete RSL definition.

1846 PRIMARY RELATION INSTANCE IS NOT IN THE ASSM.
Can only be caused by system error.

1880 COMPLEMENTARY RELATION FOR PRIMARY RELATION IS NOT IN ASSM.
Can occur due to an incomplete definition of RSL.

1920 NO ELEMENT_TYPES ARE DEFINED.

1960 NO ATTRIBUTES ARE DEFINED.

1961 NO APPLICABLE ELEMENT_TYPES ARE DEFINED.

1962 NO ATTRIBUTE LEGAL VALUES ARE DEFINED.

1980 NO RELATIONSHIPS ARE DEFINED.

1981 NO COMPLEMENTARY RELATIONSHIP IS DEFINED.

2000 NO LEGAL SUBJECTS ARE DEFINED.

2001 NO LEGAL OBJECTS ARE DEFINED.

2060 INCOMPLETE DEFINE HIERARCHY STATEMENT.

2061 DEFINE HIERARCHY STATEMENT TOO LARGE.

2062 REFERENCE TO UNDEFINED SET IN HIERARCHY DEFINITION.

2063 REFERENCE TO UNDEFINED RELATION IN DEFINE HIERARCHY.

2120 OUT OF DEFINE HIERARCHY SPACE.

Too many hierarchies have attempted to be defined during a single activation of RADX.

2121 TOO MANY START ENTRIES IN HIERARCHY.

Too many sets in the hierarchy definition have the same name as the top-of-the-hierarchy.

2122 HIERARCHY TOO COMPLEX.

There are too many connections between the entries in the hierarchy.

2180 NO MORE PROCESSING ON CURRENT PATH OF HIERARCHY.

Indicates that an error was detected while traversing a hierarchy and that the processing of the hierarchy is incomplete.

2200 TOO MANY LEVELS IN HIERARCHY.

The length of a hierarchy trace causes RADX storage of a path in the hierarchy to be exceeded.

2201 LOOP IN HIERARCHY.

Indicates that a direct or indirect loop has been found in a requirements hierarchy.

2320 LOOP IN INFORMATION NET.

Indicates that a loop was detected in a requirements hierarchy while constructing the analysis information net.

2321 PATH IN INFORMATION NET TOO LONG.

Indicates that a path in the analysis information net is too long to be stored in an internal RADX table.

2460 REPETITIVE DATA SETS (RDS) CONTAIN COMMON MEMBERS.

2540 DATA WITH USE=BETA INCLUDED IN OTHER DATA WITH USE=BETA.

2541 DATA WITH USE=GAMMA INCLUDES OTHER DATA.

2542 NO USE ATTRIBUTE.

2543 DATA SHOULD HAVE USE=BETA. VALUE ASSUMED.

2544 DATA SHOULD HAVE USE=GAMMA. VALUE ASSUMED.

2560 REFERENCE TO UNDEFINED SET IN PLOT COMMAND.

2561 PLOT DIMENSION TOO SMALL.

2562 PLOT DIMENSION TOO LARGE -- REPLACED BY MAXIMUM.

2563	REAL NUMBER ILLEGALLY SPECIFIED.
2564	ILLEGAL SYMBOL SPECIFIED IN PLOT COMMAND.
2580	ILLEGAL PERMISSION SPECIFIED.
2600	LOCALITY OF REPETITIVE DATA SET AND MEMBERS NOT THE SAME.
2620	MESSAGE CONTENTS PASSING OUTPUT_INTERFACE NEVER ASSIGNED.
2621	MESSAGE CONTENTS OF OUTPUT_INTERFACE POSSIBLY NOT ASSIGNED.
2640	ELEMENT ON CONSIDER OR IS NOT OF TYPE DATA OR ENTITY_CLASS.
2641	ELEMENT ON CONSIDER OR DOES NOT HAVE TYPE ATTRIBUTE.
2642	ELEMENT ON CONSIDER OR HAS ILLEGAL VALUE FOR TYPE ATTRIBUTE.
2643	ELEMENT ON CONSIDER OR DOES NOT HAVE RANGE ATTRIBUTE.
2644	ILLEGAL NAME IN CONDITIONAL BRANCH OR IN RANGE ATTRIBUTE.
2645	ITEM IN RANGE ATTRIBUTE EXISTS IN ASSM.
2646	ITEM IN CONDITIONAL BRANCH EXISTS IN ASSM.
2647	DUPLICATE NAME ENCOUNTERED ON CONSIDER OR NODE BRANCH.
2648	BRANCH ITEM NOT CONTAINED IN RANGE LIST ON CONSIDER OR.
2649	ALL ITEMS IN RANGE LIST NOT ENCOUNTERED ON BRANCHES.
2650	ENTITY_TYPE ON BRANCH DOES NOT EXIST IN ASSM.
2651	DUPLICATE ITEM IN RANGE LIST.
2652	ENTITY_CLASS IS NOT COMPOSED OF BRANCH ENTITY_TYPE.
2653	ALL ITEMS IN COMPOSE LIST NOT ENCOUNTERED ON BRANCHES.
2661	INFORMATION ALWAYS REASSIGNED BEFORE USED.
2662	INFORMATION SOMETIMES REASSIGNED BEFORE USED.
2663	INFORMATION SOMETIMES USED BEFORE ASSIGNED.
2664	INFORMATION ALWAYS USED BEFORE ASSIGNED.
2665	POSSIBLE USE AND ASSIGNMENT FROM DIFFERENT PARALLEL PATHS.
2666	POSSIBLE ASSIGNMENT FROM MORE THAN ONE PARALLEL PATH.
2667	USE AND ASSIGNMENT FROM DIFFERENT PARALLEL PATHS.

2668 ASSIGNMENT FROM MORE THAN ONE PARALLEL PATH.

2669 INFORMATION ASSIGNED BUT NOT USED.

2670 INFORMATION SOMETIMES ASSIGNED BUT NOT USED.

2671 SET ENTITY_TYPE WITHOUT SELECTED/CREATED ENTITY_CLASS.

2672 POSSIBLE SET ENTITY_TYPE WITHOUT SELECTED/CREATED CLASS.

2673 DESTROY ENTITY_CLASS THAT IS NOT SELECTED.

2674 POSSIBLE DESTROY ENTITY_CLASS THAT IS NOT SELECTED.

2675 DISJOINT INPUT MESSAGES REQUIRED AT THE SAME TIME.

2676 DISJOINT INPUT MESSAGES POSSIBLY REQUIRED AT THE SAME TIME.

2700 ENTITY_TYPE NOT SET FOR ENTITY_CLASS BEING UPDATED.

2720 MESSAGE NEVER FORMED WHEN OUTPUT_INTERFACE TRAVERSED.

2721 MESSAGE POSSIBLY NOT FORMED WHEN OUTPUT INTERFACE TRAVERSED.

2740 ALPHA FORMS MORE THAN ONE MESSAGE THAT PASSES SAME INTERFACE.

2760 PARTIALLY REJOINING AND-CONSTRUCT.

2780 PARTIALLY REJOINING OR-CONSTRUCT.

2800 REFERENCED SUBNET DOES NOT HAVE A STRUCTURE.

2801 TOO MANY NESTED SUBNET REFERENCES.

The number of SUBNET references is larger than that which can be processed by RADX.

2820 INFORMATION PASSING INPUT_INTERFACE NOT USED.

APPENDIX G

SIMGEN SUMMARY

G.1 SIMGEN RCL SYNTAX

Table G.1 contains a description of the syntax of SIMGEN RCL in the BNF notation described in Appendix A. For each syntax production or set of productions for the SIMGEN RCL commands, this table also identifies the number of the section in this document where the command is described.

Figure G-1 shows the syntax of SIMGEN RCL in diagrammatic form.

Table G.1 SIMGEN RCL Index

SIMGEN RCL SYNTAX	SECTION NUMBER
<SIMGEN command>::= <simulation scope declaration> | <simulation type declaration> | <simulation identification>	7.3
<simulation scope declaration>::= INCLUDE ALL [R_NETS]. | INCLUDE $\begin{bmatrix}\text{R\_NET}\\ \text{R\_NETS}\end{bmatrix} \left\{\text{R-Net-name}\right\}_1^n$. | EXCLUDE $\begin{bmatrix}\text{R\_NET}\\ \text{R\_NETS}\end{bmatrix} \left\{\text{R-Net-name}\right\}_1^n$.	7.3.1
<simulation type declaration>::= $\begin{bmatrix}\text{SIMULATION}\\ \text{SIMULATOR}\end{bmatrix}$ TYPE [IS] $\left\{\begin{matrix}\text{BETA}\\ \text{GAMMA}\end{matrix}\right\}_1^1$.	7.3.2
<simulation identification>::= $\begin{bmatrix}\text{SIMULATION}\\ \text{SIMULATOR}\end{bmatrix} \left\{\begin{matrix}\text{ID}\\ \text{IDENT}\\ \text{IDENTIFICATION}\end{matrix}\right\}_1^1$ [IS] identification-name.	7.3.3

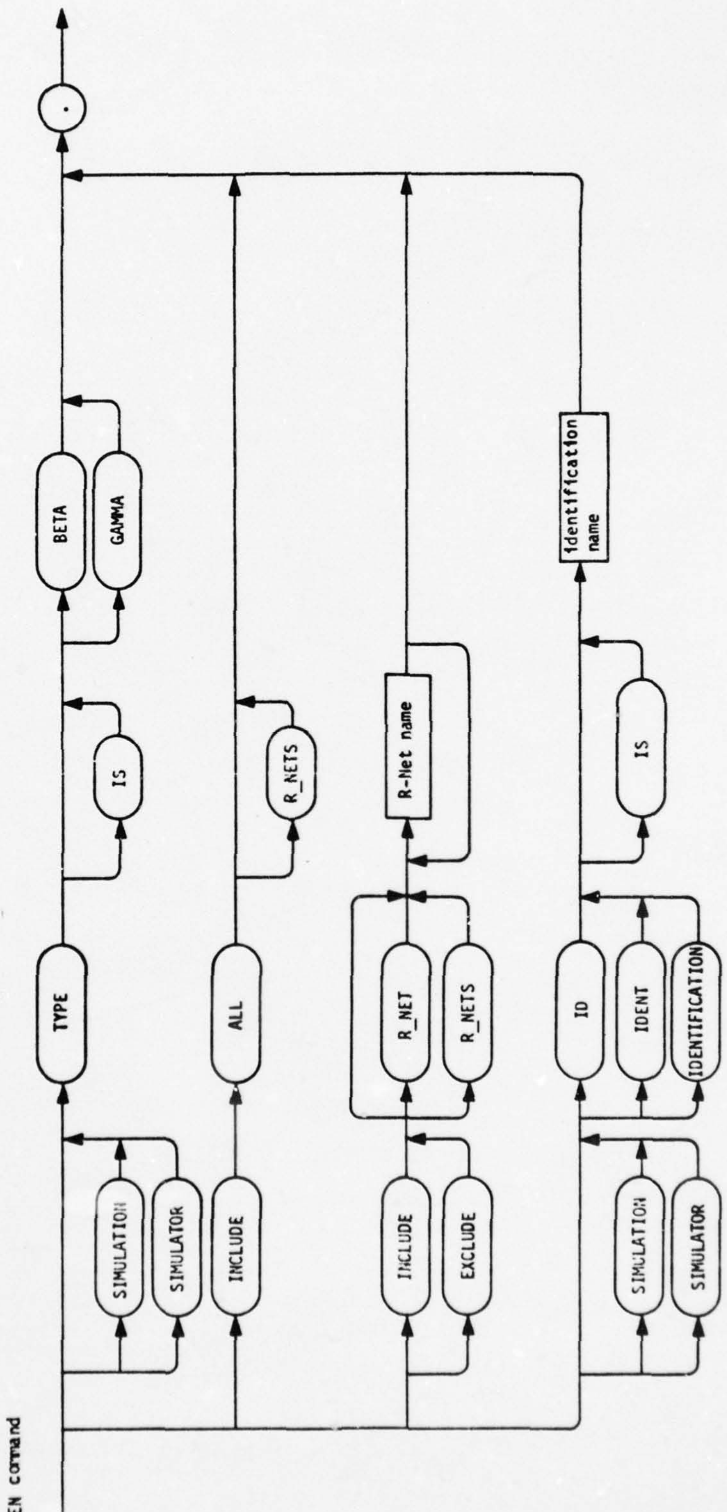
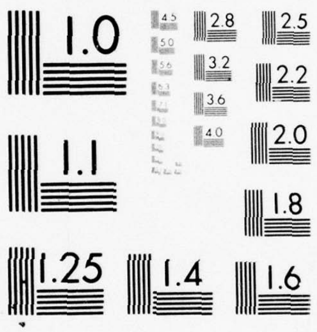


Figure G-1 SIMGEN RCL Syntax Diagram



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

G.2 BETA/GAMMA FILE ACCESS SYNTAX

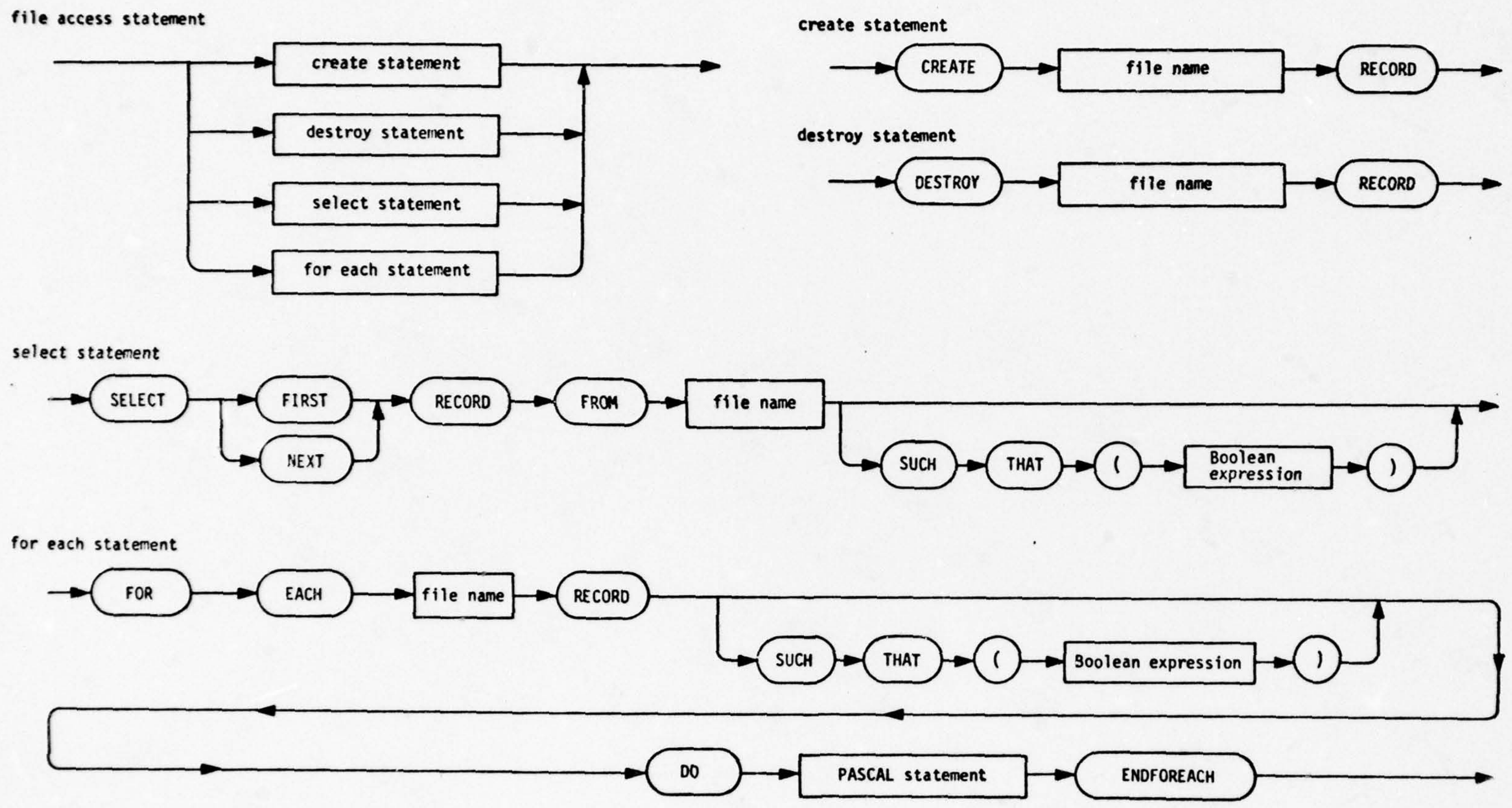
Table G.2 contains a description of the syntax of the REVS statements used to perform file operations in BETA and GAMMA executable descriptions. The syntax is presented in the BNF described in Appendix A. As noted in the table, these statements are explained in Section 7.1.1.

Figure G-2 presents the syntax of the file access statements in diagrammatic form.

Table G.2 BETA/GAMMA FILE Access Index

FILE ACCESS SYNTAX	SECTION NUMBER
<file access statement>::=	
<create statement>	
| <destroy statement>	
| <select statement>	
| <for each statement>	
<create statement>::=	
CREATE file-name RECORD	
<destroy statement>::=	
DESTROY file-name RECORD	
<select statement>::=	
SELECT $\left\{\begin{matrix}\text{FIRST}\\ \text{NEXT}\end{matrix}\right\}_{1}^{1}$ RECORD FROM file-name	
[SUCH THAT (<Boolean expression>)]	
<for each statement>::=	
FOR EACH file-name RECORD [SUCH THAT (<Boolean expression>)] DO	
<PASCAL statement> ENDFOREACH	7.1.1

NOTE: <Boolean expression> is defined in the RSL syntax presented in Appendix D. <PASCAL statement> is defined to include <file access statement> as a legal form.



NOTE: Boolean expression is defined in the RSL syntax presented in Appendix D.
Pascal statement is defined to include file access statement as a legal form.

Figure G-2 Syntax Diagrams for BETA/GAMMA FILE Access

G.3 TEST RECORDING ACCESS SYNTAX

Table G.3 contains a description of the syntax of the REVS statements used to access RECORDINGs in PERFORMANCE_REQUIREMENT TESTs. The descriptions are in the BNF described in Appendix A. As noted in the table, these statements are explained in Section 7.1.2.

Figure G-3 presents the syntax of these statements in diagrammatic form.

TABLE G.3 TEST Recording Access Index

RECORDING ACCESS SYNTAX	SECTION NUMBER
<recording access statement>::= <retrieve recording statement> | <for each recording statement> | <select recorded file statement> | <for each recorded file statement>	

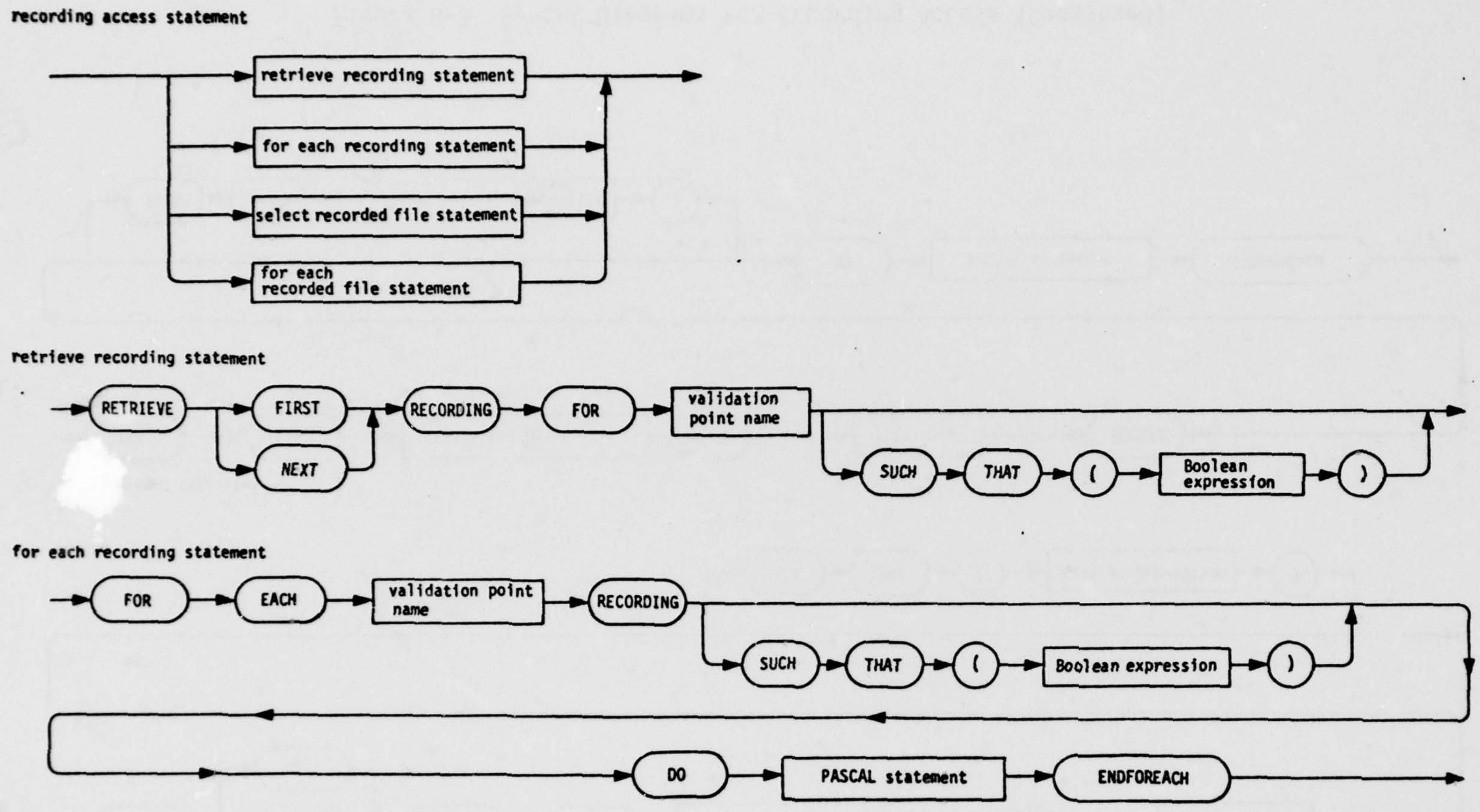
<retrieve recording statement>::=
    RETRIEVE $\left\{\begin{matrix}\text{FIRST}\\\text{NEXT}\end{matrix}\right\}_1^1$ RECORDING FOR validation-point-name
        [SUCH THAT (<Boolean expression>)]	

<for each recording statement>::=
    FOR EACH validation-point-name RECORDING
        [SUCH THAT (<Boolean expression>)]
        DO <PASCAL statement> ENDFOREACH	

<select recorded file statement>::=
    SELECT $\left\{\begin{matrix}\text{FIRST}\\\text{NEXT}\end{matrix}\right\}_1^1$ RECORD FROM validation-point-name.file-name
        [SUCH THAT (<Boolean expression>)]	

<for each recorded file statement>::=
    FOR EACH validation-point-name.file-name RECORD
        [SUCH THAT (<Boolean expression>)]
        DO <PASCAL statement> ENDFOREACH
	7.1.2

NOT: <Boolean expression> is defined in the RSL syntax presented in Appendix D. <PASCAL statement> is defined to include <recording access statement> as a legal form.


recording access statement
retrieve recording statement
for each recording statement
select recorded file statement
for each recorded file statement
retrieve recording statement
RETRIEVE
FIRST
NEXT
RECORDING
FOR
validation point name
SUCH
THAT
(
Boolean expression
)
for each recording statement
FOR
EACH
validation point name
RECORDING
SUCH
THAT
(
Boolean expression
)
DO
PASCAL statement
ENDFOREACH

NOTE: Boolean expression is defined in the RSL syntax presented in Appendix D.
Pascal statement is defined to include file access statement as a legal form.

Figure G-3 Syntax Diagrams for Recording Access

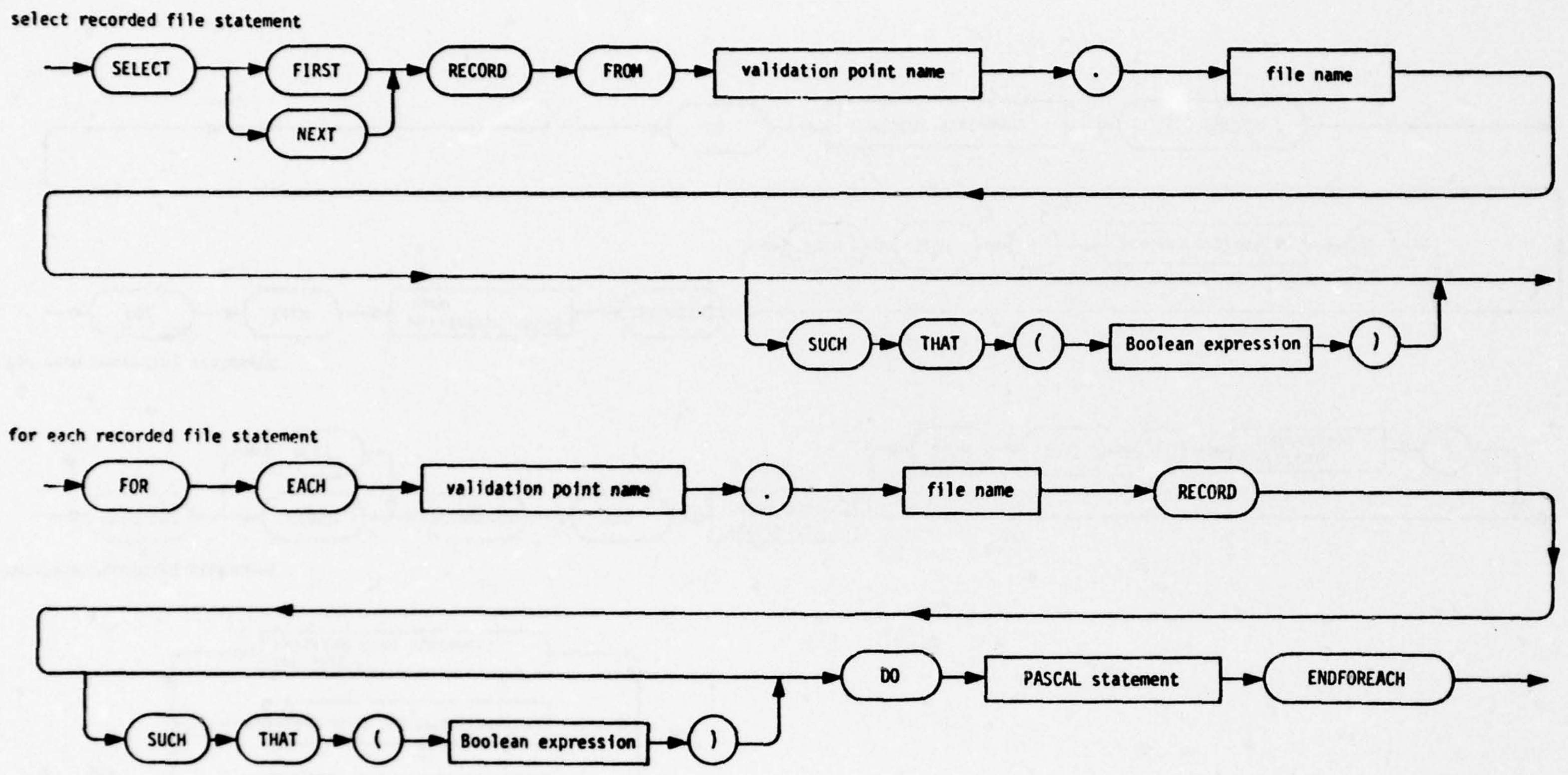


Figure G-3 Syntax Diagrams for Recording Access (Continued)

G.4 SIMGEN DIAGNOSTIC MESSAGES

SIMGEN RCL Diagnostics

The following errors are detected by SIMGEN in response to the user input RCL commands. Each of these error messages will be preceded by *ERROR* and followed on the next line by a diagnostic message of the form:

SYMBOL: last-symbol-scanned;

giving the last symbol scanned in the RCL command before detection of the error.

CONFLICT IN LIST TYPES

INCLUDE and EXCLUDE lists cannot both be used in a single generation.

ERROR IN INITIALIZATION

SIMGEN could not obtain a pointer to element type R_NET.

INPUT EMPTY

No commands have been given to enable generation of a simulation.

PERIOD MISSING

The last input statement did not contain a period terminator (non-fatal).

UNRECOGNIZABLE STATEMENT

UNRECOGNIZABLE SYMBOL

After completion of SIMGEN parsing, the following diagnostic messages may occur.

SIMULATION SCOPE (PARTIAL OR ENTIRE) NOT DEFINED.
SIMULATION GENERATION CANNOT BE PERFORMED.

SIMGEN did not receive an RCL input specifying which R_NETs were to be included in the simulation.

SIMULATION TYPE (BETA OR GAMMA) NOT SPECIFIED.
SIMULATION GENERATION CANNOT BE PERFORMED.

SIMGEN did not receive an RCL input specifying whether the simulation should be a beta type or a gamma type.

Data Base Analysis Diagnostics

Prior to translating the contents of the ASSM into executable simulation code SIMGEN executes the ANALYZE capability of RADX (see Section 6.3).

Diagnostics output by RADX in support of SIMGEN are documented in Appendix F, Section 2.

SIMGEN Translation Diagnostics

The following errors are detected by the SIMGEN function while constructing an executable simulation from the contents of the ASSM.

A FOR_EACH WITHIN A PR TEST HAS NO MATCHING ENDFOREACH

> This message will be accompanied by the name of the PERFORMANCE_REQUIREMENT.

ALPHA alpha-name HAS NO $\left\{\begin{matrix}\text{BETA}\\\text{GAMMA}\end{matrix}\right\}_1^1$.

> The BETA or GAMMA attribute for the named ALPHA cannot be found in the ASSM. SIMGEN has represented this ALPHA with a dummy procedure in the simulator program.

ASSM ERROR DETECTED AT CHECK POINT number.

> SIMGEN encountered an error in accessing the ASSM. The number indicates the point in the process at which the error occurred. Report the error to the REVS maintenance group.

ENDFOREACH DOES NOT HAVE A MATCHING FOR_EACH

> This message will be accompanied by the name of the PERFORMANCE_REQUIREMENT.

EVENT NAME NOT FOUND FOR R_NET R-Net-name.

> There is no enabling event for the R_NET.

PERFORMANCE_REQUIREMENT COULD NOT BE ACCESSED IN ASSM

> This message will appear when, through some internal system error, the PERFORMANCE_REQUIREMENT could not be accessed in the ASSM.

PERFORMANCE_REQUIREMENT TEST COULD NOT BE ACCESSED IN ASSM

> This message will appear if there is no TEST or if, through some internal system error, the TEST could not be accessed in the ASSM. The message will be accompanied by the name of the PERFORMANCE_REQUIREMENT.

SIMGEN ASSM ERROR PRINT DISABLED.

> The printing of ASSM access errors has been disabled after the detection of twenty-five errors.

SYNTAX ERROR IN PERFORMANCE REQUIREMENT TEST

This message will be accompanied by the name of the PERFORMANCE_ REQUIREMENT and the name of the function being translated. In addition the name of the VALIDATION_POINT, the name of the VALIDATION_POINT FILE, and the number of the word in error will appear if known and applicable.

TEST IS NOT IN THE ASSM OR IS NOT AN ATTRIBUTE

This message will appear when the attribute TEST has been deleted from the language or when TEST is no longer an attribute.

$\left\{\begin{matrix}\text{CREATE}\\\text{DESTROY}\\\text{FOR EACH}\\\text{SELECT}\end{matrix}\right\}_1^1$ STATEMENT IN *alpha-name* CONTAINS AN ERROR: OPERAND
identifier NOT FOUND IN DATA BASE.

A statement in the BETA/GAMMA text of the indicated ALPHA has an operand that is not in the ASSM. The statement is treated as PASCAL.

$\left\{\begin{matrix}\text{CREATE}\\\text{DESTROY}\\\text{FOR EACH}\\\text{SELECT}\end{matrix}\right\}_1^1$ STATEMENT IN *alpha-name* CONTAINS SYNTAX ERROR.

A statement in the BETA/GAMMA text of the indicated ALPHA is not consistent with the syntax defined in Section 7.1.1.

identifier IS ILLEGAL OPERAND OF*$\left\{\begin{matrix}\text{CREATE}\\\text{DESTROY}\\\text{FOR EACH}\\\text{SELECT}\end{matrix}\right\}_1^1$* STATEMENT IN *alpha-name*.

A statement in the BETA/GAMMA text of the indicated ALPHA has an operand that is not of type FILE. The statement is treated as PASCAL.

identifier IS $\left\{\begin{matrix}\text{INPUT TO}\\\text{OUTPUT BY}\end{matrix}\right\}_1^1$ *alpha-name*.
NOT CONSISTENT WITH SPECIFIED REQUIREMENTS.

SIMGEN has detected the indicated relationship in the BETA/GAMMA text of the named ALPHA. The relationship is not included in the requirements that have been specified for this ALPHA.

identifier WAS NOT $\left\{\begin{matrix}\text{USED IN}\\\text{OUTPUT BY}\end{matrix}\right\}_1^1$ *alpha-name*.

For the indicated ALPHA, there is an INPUTS or OUTPUTS relationship, with the identifier as the object, that is not implemented in the BETA/GAMMA text.

OPERAND OF *$\left\{\begin{matrix}\text{CREATE}\\\text{DESTROY}\end{matrix}\right\}_1^1$* NOT FOUND IN *alpha-name*.

$\left\{\begin{matrix}\text{CREATE}\\\text{DESTROY}\end{matrix}\right\}_1^1$ DELETED FROM ALPHA TEXT.

There is a syntax error in a CREATE or DESTROY statement in the BETA/GAMMA text of tne indicated ALPHA. SIMGEN was unable to find the operand of the statement, and the operator (e.g., CREATE) was not translated.

0342 DATA-ITEM ENUMERATED TYPE HAS NO ATTRIBUTE OF RANGE

A data element has the attribute TYPE with a value of ENUMERATION but the data element does not have the attribute RANGE.

0343 DATA-ITEM HAS ENUMERATED TYPE BUT NO RANGE ATTRIBUTE AADBPTR

A data element has the attribute TYPE with a value of ENUMERATION but the definition of the attribute RANGE cannot be located in the ASSM. Either it has been deleted from the language, RANGE is defined in the ASSM but not as an attribute, or there has been a system/hardware error.

0344 DATA-ITEM HAS AN UNKNOWN TYPE ATTRIBUTE STRING

A data element has the attribute TYPE but the value is not REAL, INTEGER, BOOLEAN, or ENUMERATION.

0345 DATA-ITEM TYPE ATTRIBUTE HAS A BAD STRING SEGMENT

A data element has the attribute TYPE but the value could not be accessed from the ASSM due to a system/hardware error.

0346 DATA-ITEM HAS NO ATTRIBUTE OF TYPE

A data element does not have the attribute TYPE.

0347 NO TYPE ATTRIBUTE AADBPTR

The definition of the attribute TYPE could not be located in the ASSM. Either it has been deleted from the language, TYPE is defined in the ASSM but not as an attribute, or there has been a system/hardware error.

0521 DATA-ITEM INITIAL VALUE HAS A BAD STRING SEGMENT

The value of the attribute INITIAL_VALUE for a data element could not be accessed due to a faulty ASSM.

0523 NO INITIAL VALUE ATTRIBUTE AADBPTR

The definition of the attribute INITIAL_VALUE could not be located in the ASSM. Either it has been deleted from the language, INITIAL_VALUE is defined in the ASSM but not as an attribute, or there has been a system/hardware error.

0681 LOCALITY IS NOT AN ATTRIBUTE

LOCALITY is defined in the ASSM but not as an attribute.

0682 LOCALITY IS NOT IN THE ASSM

The definition of LOCALITY has been deleted from the language.

0683 TYPE IS NOT AN ATTRIBUTE

TYPE is defined in the ASSM but not as an attribute.

0684 TYPE IS NOT IN THE ASSM

The definition of TYPE has been deleted from the language.

0685 RANGE IS NOT AN ATTRIBUTE

RANGE is defined in the ASSM but not as an attribute.

0686 RANGE IS NOT IN THE ASSM

The definition of RANGE has been deleted from the language.

0687 INITIAL_VALUE IS NOT AN ATTRIBUTE

INITIAL_VALUE is defined in the ASSM but not as an attribute.

0688 INITIAL_VALUE IS NOT IN THE ASSM

The definition has been deleted from the language.

0701 FILE NAME STRING IS BAD

The name of a FILE element could not be accessed due to a system/hardware error.

0741 MESSAGE NAME STRING IS BAD

The name of a MESSAGE element could not be accessed due to a system/hardware error.

0742 INTERFACE NAME STRING IS BAD

The name of an INTERFACE element could not be accessed due to a system/hardware error.

0801 FILE LOCALITY ATTRIBUTE STRING IS BAD

The value of the attribute LOCALITY of a FILE element could not be accessed due to a system/hardware error.

0802 ATTRIBUTE LOCALITY IS ABSENT

The definition of the attribute LOCALITY could not be located in the ASSM. Either it has been deleted from the language, LOCALITY is defined in the ASSM but not as an attribute, or there has been a system/hardware error.

0821 ENUMERATED TYPE VALUE STRING HAS MORE THAN AASTRLEN CHARCTRS

The length (in characters) of an enumeration value name is more than 60 characters. (See the definition of the attribute RANGE.)

0822 ENUMERATED TYPE RANGE SEGMENT IS BAD

The value of the attribute RANGE could not be accessed due to a system/hardware error.

0823 ENUMERATED TYPE ENDED UNEXPECTEDLY (NO DOUBLE QUOTE)

The syntax specifying the value of the attribute RANGE is incorrect indicating a system/hardware error.

0824 ENUMERATED TYPE DOES NOT BEGIN WITH DOUBLE QUOTE

The syntax specifying the value of the attribute RANGE is incorrect indicating a system/hardware error.

0825 ENUMERATED TYPE HAS AN EMPTY (ZERO-LENGTH) VALUE

The syntax specifying the value of the attribute RANGE is incorrect indicating a system/hardware error.

0841 CLASS DATA ITEM NAME STRING IS BAD

The name of a DATA element could not be accessed due to a system/hardware error.

0842 ENTITY_TYPE NAME STRING IS BAD

The name of an ENTITY_TYPE element could not be accessed due to a system/hardware error.

0843 ENTITY_CLASS NAME STRING IS BAD

The name of an ENTITY_CLASS element could not be accessed due to system/hardware error.

0844 CLASS FILE NAME STRING IS BAD

The name of a FILE element could not be accessed due to a system/hardware error.

0861 INSTANCE FILE NAME STRING IS BAD

The name of a FILE element could not be accessed due to a system/hardware error.

0881 INSTANCE DATA-ITEM NAME STRING IS BAD

The name of a DATA element could not be accessed due to a system/hardware error.

0941 SIMPLE DATA LOCALITY STRING IS BAD

The value of the attribute LOCALITY could not be accessed due to a system/hardware error.

0942 ATTRIBUTE LOCALITY IS ABSENT

The definition of the attribute LOCALITY could not be located in the ASSM. Either it has been deleted from the language, LOCALITY is defined in the ASSM but not as an attribute, or there has been a system/hardware error.

0343 SIMPLE DATA NAME STRING IS BAD

The name of a data element could not be accessed due to a system/hardware error.

5000 VALIDATION POINT NAME STRING CANNOT BE ACCESSED IN ASSM

The name of a VALIDATION_POINT could not be accessed due to a system/hardware error.

APPENDIX H
SIMXQT SUMMARY

H.1 SIMXQT RCL SYNTAX

Table H.1 contains a description of the syntax of SIMXQT RCL in the BNF notation described in Appendix A. For each syntax production or set of productions for the SIMXQT RCL commands, this table also identifies the number of the section in this document where the command is described.

Figure H-1 shows the syntax of SIMXQT RCL in diagrammatic form.

Table H.1 SIMXQT RCL Index

SIMXQT RCL SYNTAX	SECTION NUMBER

<SIMXQT command>::=
    <start time declaration>
  | <end time declaration>
  | <simulation run identification>
	7.4

<start time declaration>::= $\begin{bmatrix}\text{SIMULATION}\\\text{SIMULATOR}\end{bmatrix}$ START [TIME] = real-number.	7.4.1
<end time declaration>::= $\begin{bmatrix}\text{SIMULATION}\\\text{SIMULATOR}\end{bmatrix}$ END [TIME] = real-number.	7.4.1
<simulation run identification>::= $\begin{bmatrix}\text{SIMULATION}\\\text{SIMULATOR}\end{bmatrix}$ RUN $\left\{\begin{matrix}\text{ID}\\\text{IDENT}\\\text{IDENTIFICATION}\end{matrix}\right\}_1^1$ [IS] identification-name.	7.4.2

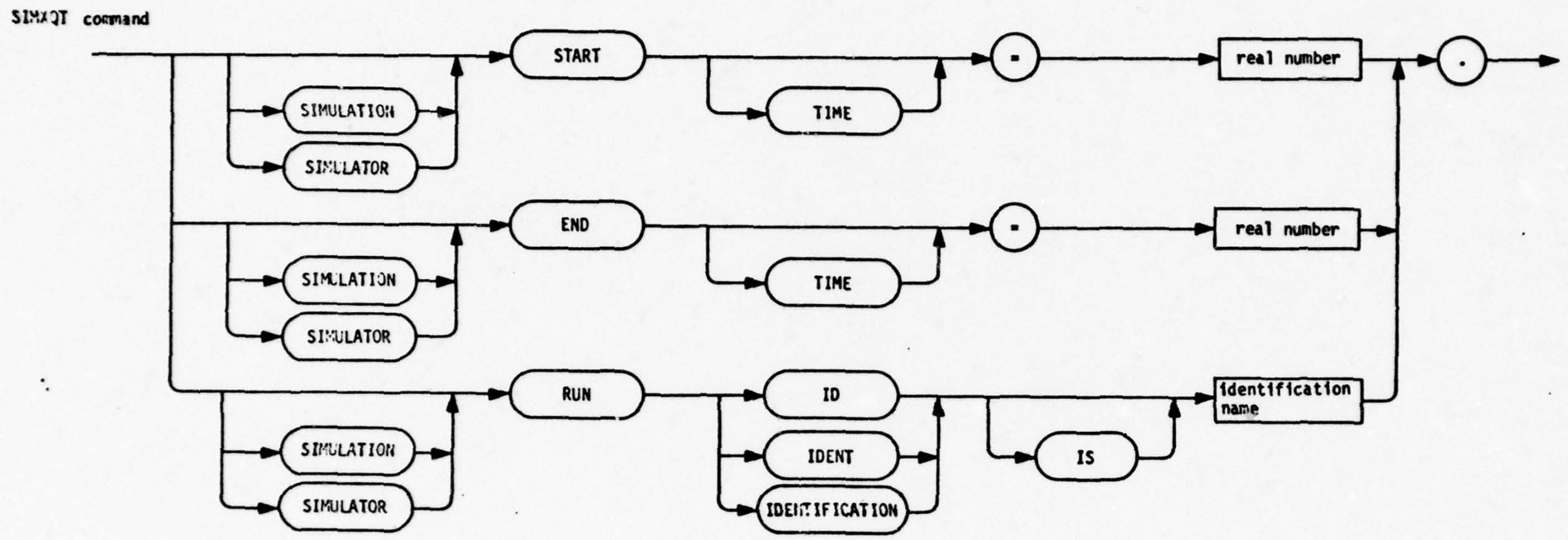


Figure H-1 SIMXQT RCL Syntax Diagram

H.2 SIMXQT DIAGNOSTIC MESSAGES

The following errors are detected by SIMXQT in response to the user input RCL commands. Each of these error messages will be preceded by *ERROR* and followed on the next line by a diagnostic message of the form:

```
SYMBOL: last-symbol-scanned;
```

giving the last symbol scanned in the RCL command before detection of the error.

ERROR IN REAL NUMBER

INPUT EMPTY

PERIOD MISSING

TIME MUST BE LARGER THAN -1.0E4

UNRECOGNIZABLE STATEMENT

UNRECOGNIZABLE SYMBOL

The execution of the simulation requires the input of both a start and end time, with the end time greater than the start time. If these inputs have not been correctly provided, the following diagnostics will be produced:

END TIME NOT SPECIFIED

START TIME NOT SPECIFIED

START TIME >= END TIME

H.3 SIMULATOR PROGRAM DIAGNOSTIC MESSAGES

Initialization Diagnostics

During the initialization phase of a simulator program execution, the user supplied start and end times are read into the simulator program from a file (the EESUIF) constructed by the SIMXQT function. The following diagnostics may occur:

START TIME NOT INPUT

No start time was found on the EESUIF.

END TIME NOT INPUT

No end time found on the EESUIF.

The simulator program initialization also reads another file, the EEDF, constructed by SIMGEN. In reading this file, simulator initialization will verify that it is the correct file. The following diagnostics may occur:

NO EEDF FILE PROVIDED

No EEDF file is available.

EEDF FILE DOES NOT MATCH THE SIMULATION PROGRAM TEXT

The EEDF file read in was not created by the same SIMGEN execution as was the simulator program.

If any of the above errors was detected during initialization, the following diagnostic will be output and the simulator program will not be executed.

SIMULATION PROGRAM CANNOT BE EXECUTED

Execution Diagnostics

ATTEMPTED BACKUP IN TIME, EVENT IGNORED, TIME = simulation-time

ATTEMPTED CAUSE OF AN UNKNOWN EVENT, EVENT IGNORED, TIME = simulation-time

EVENT NAME IS event-name

An attempt has been made to cause an event which is not known to the simulator event manager. The event will be ignored.

EVENT NAME IS event-name EVENT TIME = desired-event-time

An attempt has been made to cause an event to occur at an earlier time than current simulation time. The event is ignored.

VALIDATION POINT = xxxx
NO OWNER SELECTED FOR $\left\{\begin{matrix}\text{data-name}\\ \text{file-name}\end{matrix}\right\}_1^1$.

A VALIDATION_POINT attempted to RECORD DATA which is either CONTAINED in a FILE or ASSOCIATED with an ENTITY_TYPE or ENTITY_CLASS and the recording was attempted without a FILE record or entity having been previously selected; or the VALIDATION_POINT attempted to RECORD a FILE ASSOCIATED with an ENTITY_TYPE or ENTITY_CLASS without an entity having been selected. This message will be preceded by a message identifying the name of the VALIDATION_POINT. (Note that xxxx is the ordinal of the VALIDATION_POINT at which recording was attempted.)

4001 NO SET TYPE FOR A NEW ENTITY_CLASS INSTANCE

An instance of an entity-class was created and is now being de-selected but a SET operation has not been done to establish the type.

4021 DATA-SET INSTANCE STATUS IS OLD BUT NO INSTANCE PRESENT

A system internal error occurring when an entity or FILE record to be de-selected is marked as existing but the pointer to it is NIL.

4061 SELECT NEXT WITH NO SELECT FIRST OR WITH NO FOUND CHECK

A SELECT with a NEXT option has encountered a "no current instance" condition. This may occur if a SELECT with a FIRST option has not preceded it or if the global data element RECORD_FOUND has not been monitored properly to determine that no instance is selected.

4201 DATA-SET HEADER BEING LOADED INTO FROM INSTANCE IS NOT EMPTY

An error detected on the selection of an entity which ASSOCIATES a FILE; there is already a FILE in existence which is not attached to an entity. This can occur by creating a FILE without having previously selected or created the entity and then performing a SELECT or FOR EACH on the ENTITY_CLASS or ENTITY_TYPE.

4241 DATA-SET BEING COPIED INTO IS NOT EMPTY

A system internal error occurring on the establishment of a MESSAGE.

4281 INSTANCE TO BE DESTROYED IS LOCKED BY A FOR-EACH

A DESTROYS ENTITY_CLASS has been encountered inside of a FOR EACH operation on a FILE ASSOCIATED with the currently selected entity of the ENTITY_CLASS.

4301 DATA-SET HEADER TO BE DESTROYED IS NOT EMPTY

A system internal error occurring on the destruction of a MESSAGE or ENTITY_TYPE (or class) which MAKE a FILE or ASSOCIATES a FILE, respectively. The condition is abnormal because the FILE should have been destroyed earlier.

4341 ATTEMPT TO SAVE ENTITY_CLASS WITH NO ENTITY_TYPE SELECTED

A system internal error occurring on the save portion of a FOR EACH operation.

4342 ATTEMPT TO SAVE ENTITY_TYPE WHICH IS NOT SELECTED

A system internal error occurring on the save portion of a FOR EACH operation.

4343 ATTEMPT TO SAVE DATA-SET WHICH HAS NO VALID INSTANCE SELECTED

A system internal error occurring on the save portion of a FOR EACH operation.

4361 ATTEMPT TO RESTORE ENTITY_TYPE WHICH IS NOT IN ENTITY_CLASS

A system internal error occurring on the restore portion of a FOR EACH operation.

4362 ATTEMPT TO RESTORE DATA-SET WHICH WAS NOT SAVED

A system internal error occurring on the restore portion of a FOR EACH operation.

4363 ATTEMPT TO RESTORE A NON-EXISTENT INSTANCE

A system internal error occurring on the restore portion of a FOR EACH operation.

4401 DESTROY ENTITY_CLASS WITH NO ENTITY SELECTED

An error occurring on the destruction of an instance of an entity-class when no instance is selected.

4402 DESTROY DATA-SET WITH NO INSTANCE SELECTED

A DESTROY operation on a FILE has been encountered when no record is selected in the FILE.

4441 SET TYPE ON ENTITY_CLASS

A system internal error - SETS has been attempted on an ENTITY_CLASS.

4442 SET TYPE ON FILE OR INTERFACE

A system internal error - SETS has been attempted on a FILE or interface.

4443 SET TYPE WITH NO NEW OR SELECTED ENTITY

An error occurring when a SETS ENTITY_TYPE operation is done without an entity being selected.

4444 SET TYPE WITH AN INVALID INSTANCE SELECTED

An error occurring when a SETS ENTITY_TYPE operation is done without an entity being selected.

4501 ATTEMPT TO TERMINATE A NON-OUTPUT-INTERFACE

A system internal error occurring when a data-set which is not an interface is terminated.

4502 A MESSAGE WAS NOT FORMED ON AN OUTPUT_INTERFACE

An error occurring when an OUTPUT_INTERFACE is encountered without a MESSAGE having been FORMed for the interface.

4541 DATA-SET TYPE INCONSISTENT WITH CURRENT INSTANCE POINTER

A system internal error occurring on the restore portion of a VALIDATION_POINT execution. The data-set (or specifically - file) name saved during the save portion of the VALIDATION_POINT execution does not agree with the data-set name passed to the restore function. This is the first of two different cross checks to guarantee restoration of the same data-set as that saved.

4542 CURRENT STATIC DATA POINTER IS NIL

A system internal error occurring on the restore portion of a VALIDATION_POINT execution. The pointer to a dynamically allocated record of the file's original environment is NIL. This condition is abnormal because the record is allocated and the pointer set during the save portion of a VALIDATION_POINT execution (which precedes the restore).

4543 DATA_SET TYPE INCONSISTENT WITH STATIC DATA POINTER

A system internal error occurring on the restore portion of a VALIDATION_POINT execution. The data-set (or specifically - file) name saved during the save portion of the VALIDATION_POINT execution does not agree with the data-set name passed to the restore function. This is the second of two different cross checks to guarantee restoration of the same data-set as that saved.

APPENDIX I

SIMDA SUMMARY

I.1 SIMDA RCL SYNTAX

Table I.1 contains a description of the syntax of the SIMDA RCL command in the BNF notation described in Appendix A. The section number appearing in the table identifies the section of this document where the command is described.

Figure I-1 shows the SIMDA RCL syntax in diagrammatic form.

Table I.1 SIMDA RCL Index

SIMDA RCL SYNTAX	SECTION NUMBER
<SIMDA command>::=	
TEST ALL $\left[\begin{array}{l}\text{PERFORMANCE\_REQUIREMENT}\\ \text{PERFORMANCE\_REQUIREMENTS}\end{array}\right]$.	
$\mid$ TEST ALL EXCEPT $\left[\begin{array}{l}\text{PERFORMANCE\_REQUIREMENT}\\ \text{PERFORMANCE\_REQUIREMENTS}\end{array}\right]$ $\left\{\text{performance-requirement-name}\right\}_1^n$.	
$\mid$ TEST $\left[\begin{array}{l}\text{PERFORMANCE\_REQUIREMENT}\\ \text{PERFORMANCE\_REQUIREMENTS}\end{array}\right]$ $\left\{\text{performance-requirement-name}\right\}_1^n$.	7.5

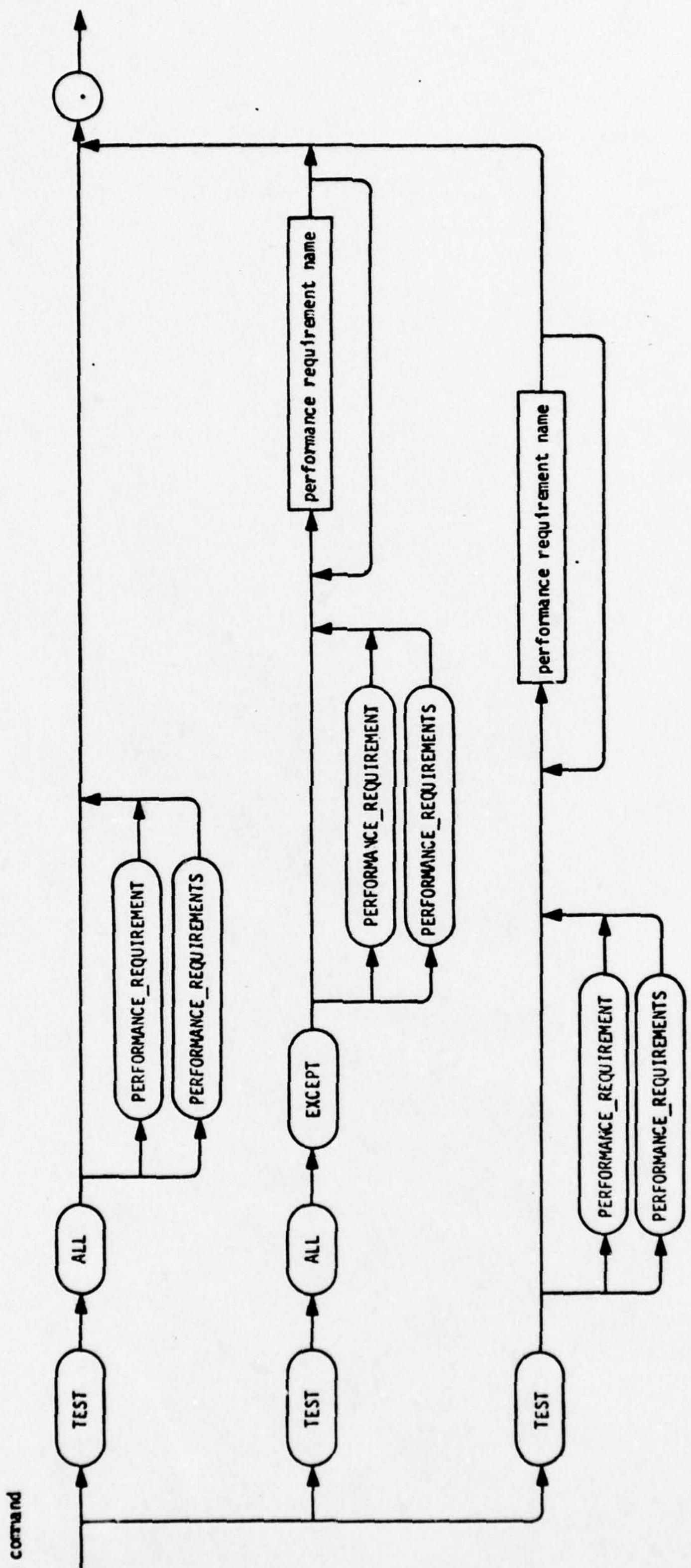


Figure I-1 SIMDA RCL Syntax Diagram

I.2 SIMDA DIAGNOSTIC MESSAGES

The Simulation Data Analysis Function (SIMDA) must read in a file, the EEDF, created by SIMGEN to obtain the names of PERFORMANCE_REQUIREMENTs included in the simulation. If the EEDF file is missing or contains no PERFORMANCE_REQUIREMENT names, then SIMDA will output the following diagnostic and will terminate.

```
NO PERFORMANCE_REQUIREMENTS AVAILABLE FOR TESTING
```

The following errors are detected by SIMDA in response to the user input RCL commands. Each of these error messages will be preceded by *ERROR* and followed on the next line by a diagnostic message of the form:

```
SYMBOL:  last-symbol-scanned.
```

giving the last symbol scanned before detection of the error.

CONFLICT IN LIST TYPES

Only one form of the TEST list may be given. If ALL or ALL EXCEPT was used, no other list may be specified.

INPUT EMPTY

PERIOD MISSING

The last input statement did not contain a terminating period. This error does not affect the processing of that statement.

UNRECOGNIZABLE STATEMENT

UNRECOGNIZABLE SYMBOL

I.3 SIMULATION POST PROCESSOR DIAGNOSTIC MESSAGES

Initialization Diagnostics

During the initialization phase of a simulation post processor execution, the recording data base and the validation information file are correlated with the simulator post processor program to guarantee compatibility. If some irregularity is discovered during this check, one of the following messages will be issued.

VP RECORDING DATABASE COULD NOT BE OPENED

> This message will appear when through some internal system or user error the recording data base could not be opened.

VP RECORDING DATABASE HEADER IS ABSENT

> This message will appear when through some internal system or user error the recording data base identification data is missing.

RECORDING DOES NOT MATCH TEST

> This message will appear when the recording data base input to the simulation post processor program was not generated by the simulator which matches the simulation post processor program (i.e., both were created by Simulation Generation at the same time). This message will be accompanied by the time, date, and ID of creation for both the recording data base and the simulation post processor program.

VALIDATION INFORMATION FILE (VVIF) IS EMPTY

> This message will appear when no validation information file has been supplied.

VALIDATION_INFORMATION_FILE HEADER DOES NOT AGREE WITH TEST

> This message will appear when the validation information file input to the simulation post processor program was not generated from the simulator which matches the simulation post processor program.

NO VALIDATION POINT INSTANCES RECORDED IN DATABASE

> This message will appear when no DATA was recorded at any VALIDATION_POINT during the simulator execution.

VALIDATION_POINT xxxx HAS NO RECORDINGS IN THE DATABASE

> This message will appear when no data was recorded at validation point xxxx (where xxxx is the validation point ordinal) during the simulator execution. In the post processor, there is a retrieval procedure corresponding to each VALIDATION_POINT in the simulator; the name of the procedure is VVxxxx where xxxx is the validation point ordinal. The RSL name of the VALIDATION_POINT appears in a comment directly following the retrieval procedure heading.

Execution Diagnostics

PERFORMANCE REQUIREMENT performance-requirement-name NOT FOUND

This message will appear when the user has requested execution of a performance requirement test that could not be found.

APPENDIX J

RSLXTND SUMMARY

J.1 RSL EXTENSION SYNTAX

Table J.1 contains a description of the syntax of RSL Extension commands in the BNF notation described in Appendix A. For each syntax production or set of productions for the RSL Extension commands, this table also identifies the number of the section in this document where the command is described. Note that an RSL command (the syntax for which is summarized in Appendix D) is a legal command to the RSL Extension (RSLXTND) function.

Figure J.1 shows the syntax of RSL Extension commands in diagrammatic form.

J.2 RSLXTND DIAGNOSTIC MESSAGES

The diagnostic messages output by the RSLXTND function are the same as those output by the RSL function and are documented in Section 5 of Appendix D.

Table J.1 RSL Extension Index

RSL EXTENSION SYNTAX	SECTION NUMBER
<RSL Extension command>::= <extension control command> | <new element type definition> | <element type modification> | <element type deletion> | <new attribute definition> | <attribute modification> | <attribute deletion> | <new relation definition> | <relation modification> | <relation deletion> | <RSL command	8.0
<extension control command>::= IDENTIFICATION name. | EXTENSION_PERMISSION name. | CONTROL_PERMISSION name. | RESCIND PERMISSION name.	8.1
<new element type definition>::= [DEFINE] ELEMENT_TYPE element-type-name comment. $\left\{\text{[INSERT] <structure applicability declaration>}\right\}_0^n$.	
<structure applicability declaration>::= STRUCTURE APPLICABILITY $\left\{\begin{matrix}\text{NET}\\\text{PATH}\end{matrix}\right\}_1^1$.	8.2.1
<element type modification>::= [MODIFY] ELEMENT_TYPE element-type-name [comment]. $\left\{\left[\begin{matrix}\text{INSERT}\\\text{REMOVE}\end{matrix}\right]\text{<structure applicability declaration>}\right\}_0^n$	8.3.1
<element type deletion>::= DELETE ELEMENT_TYPE element-type-name.	8.4.1
<new attribute definition>::= [DEFINE] ATTRIBUTE attribute-name comment. $\left\{\text{[INSERT] <attribute definition sentence>}\right\}_0^n$	
<attribute definition sentence>::= <applicable type declaration> | <legal value declaration>	
<applicable type declaration>::= APPLICABLE [ELEMENT_TYPE] <element types>.	
<element types>::= ALL | [ALL EXCEPT] $\left\{\text{element-type-name}\right\}_1^n$	
<legal value declaration>::= VALUE $\left\{\begin{matrix}\text{NUMERIC}\\\text{TEXT}\\\text{NAMED}\\\text{value-name}\end{matrix}\right\}_1^1$ [comment].	8.2.2

Table J.1 RSL Extension Index (Continued)

RSL EXTENSION SYNTAX	SECTION NUMBER
<attribute modification>::= [MODIFY] ATTRIBUTE attribute-name [comment]. $\left\{\begin{array}{l}\text{[INSERT] <attribute definition sentence>}\\ \text{<applicable type declaration removal>}\\ \text{<legal value declaration removal>}\end{array}\right\}_0^n$	
<applicable type declaration removal>::= REMOVE APPLICABLE [ELEMENT_TYPE] $\left\{\begin{array}{l}\text{ALL}\\ \left\{\text{element-type-name}\right\}_1^n\end{array}\right\}_1^1$.	
<legal value declaration removal>::= REMOVE VALUE $\left\{\begin{array}{l}\text{NUMERIC}\\ \text{TEXT}\\ \text{NAMED}\\ \text{value-name}\end{array}\right\}_1^1$.	8.3.2
<attribute deletion>::= DELETE ATTRIBUTE attribute-name.	8.4.2
<new relation definition>::= [DEFINE] $\left\{\begin{array}{l}\text{RELATION}\\ \text{RELATIONSHIP}\end{array}\right\}_1^1$ relation-name [("relation-optional-word")] comment. $\left\{\text{[INSERT] <relation definition sentence>}\right\}_0^n$	
<relation definition sentence>::= <complementary relation declaration> | <subject type declaration> | <object type declaration>	
<complementary relation declaration>::= COMPLEMENTARY $\left\{\begin{array}{l}\text{RELATION}\\ \text{RELATIONSHIP}\end{array}\right\}_1^1$ relation-name [("relation-optional-word")].	
<subject type declaration>::= SUBJECT [ELEMENT_TYPE] <element types>.	
<object type declaration>::= OBJECT [ELEMENT_TYPE] <element types>.	8.2.3
<relation modification>::= [MODIFY] $\left\{\begin{array}{l}\text{RELATION}\\ \text{RELATIONSHIP}\end{array}\right\}_1^1$ relation-name [("relation-optional-word")] [comment]. $\left\{\begin{array}{l}\text{[INSERT] <relation definition sentence>}\\ \text{<complementary relation declaration removal>}\\ \text{<subject type declaration removal>}\\ \text{<object type declaration removal>}\end{array}\right\}_0^n$	
<complementary relation declaration removal>::= REMOVE COMPLEMENTARY $\left\{\begin{array}{l}\text{RELATION}\\ \text{RELATIONSHIP}\end{array}\right\}$ relation-name [("relation-optional-word")]	
<subject type declaration removal>::= REMOVE SUBJECT [ELEMENT_TYPE] $\left[\begin{array}{l}\text{ALL}\\ \left\{\text{element type name}\right\}_1^n\end{array}\right]$.	
<object type declaration removal>::= REMOVE OBJECT [ELEMENT_TYPE] $\left[\begin{array}{l}\text{ALL}\\ \left\{\text{element type name}\right\}_1^n\end{array}\right]$.	8.3.3
<relation deletion>::= DELETE $\left\{\begin{array}{l}\text{RELATION}\\ \text{RELATIONSHIP}\end{array}\right\}_1^1$ relation-name [("relation-optional-word")].	8.4.3

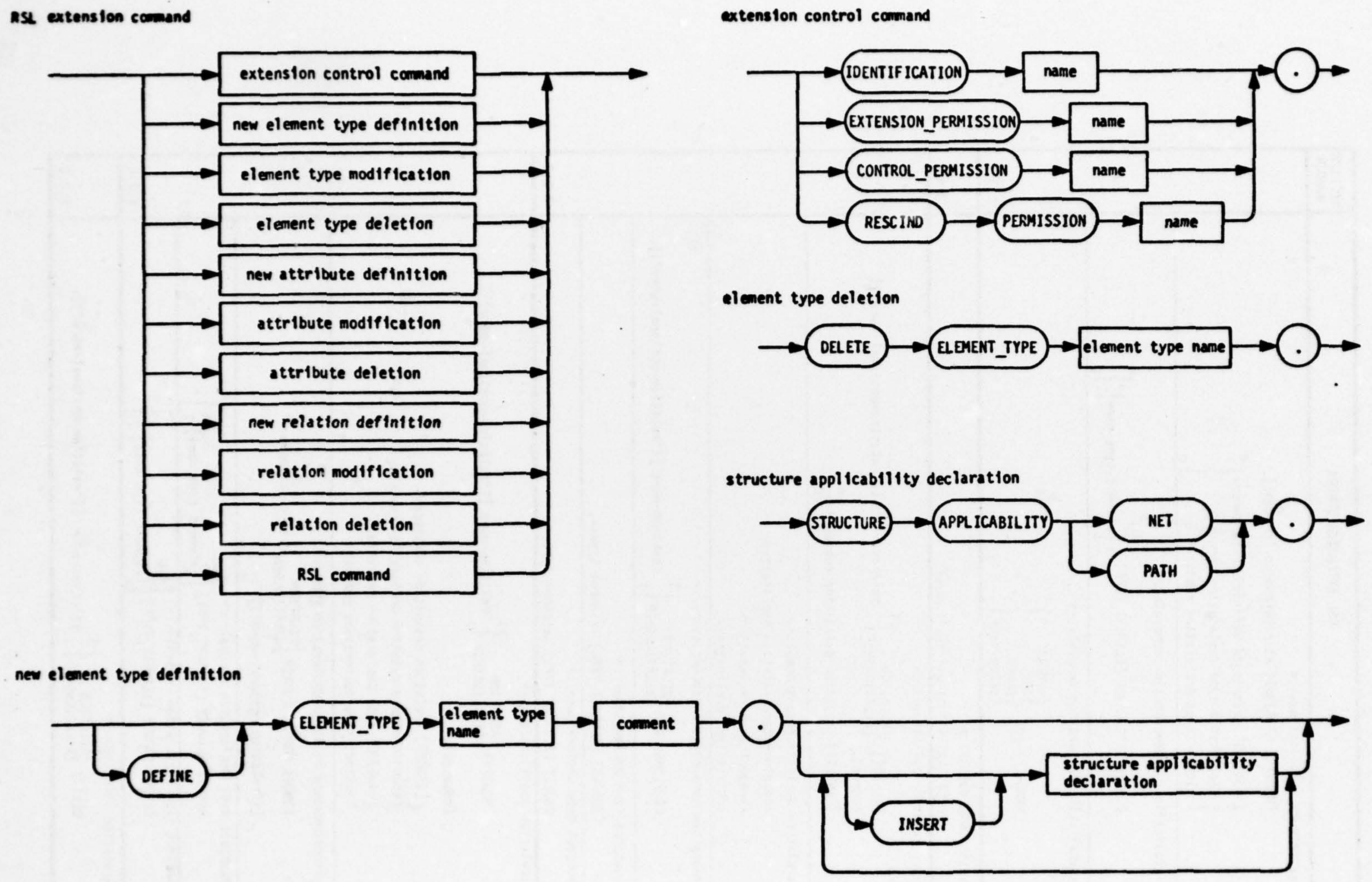
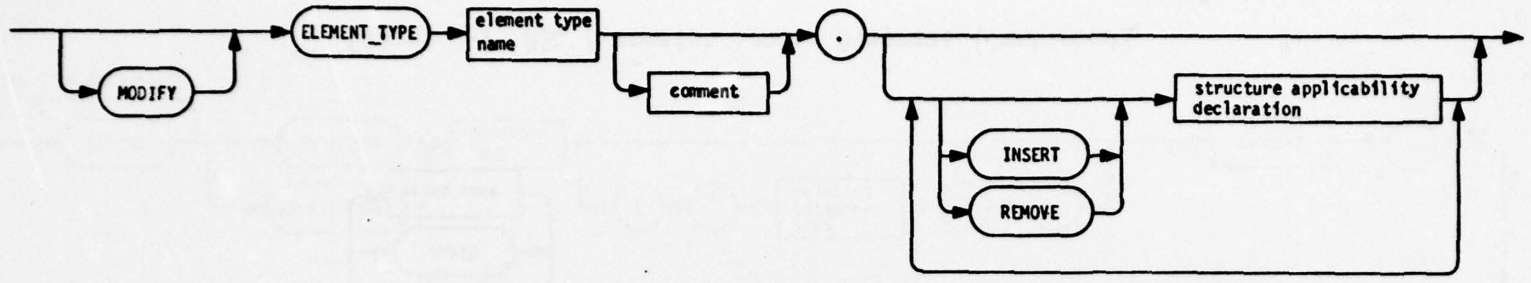
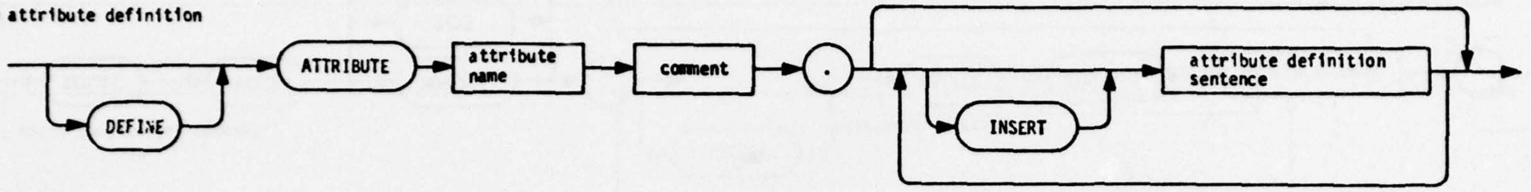


Figure J-1 RSL Extension Syntax Diagrams element type modification



new attribute definition



attribute definition sentence

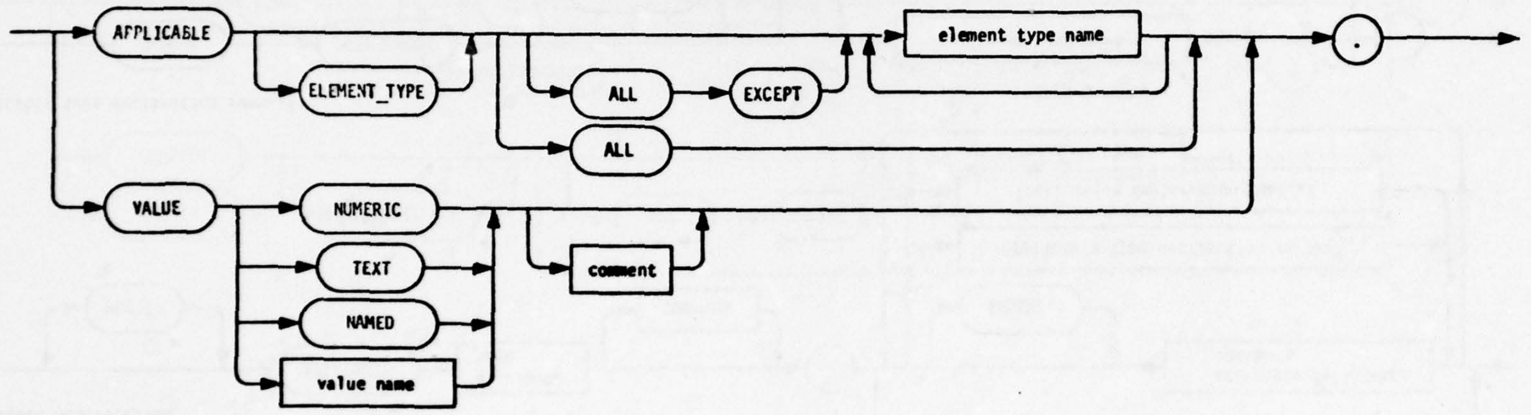


Figure J-1 RSL Extension Syntax Diagrams (Continued)

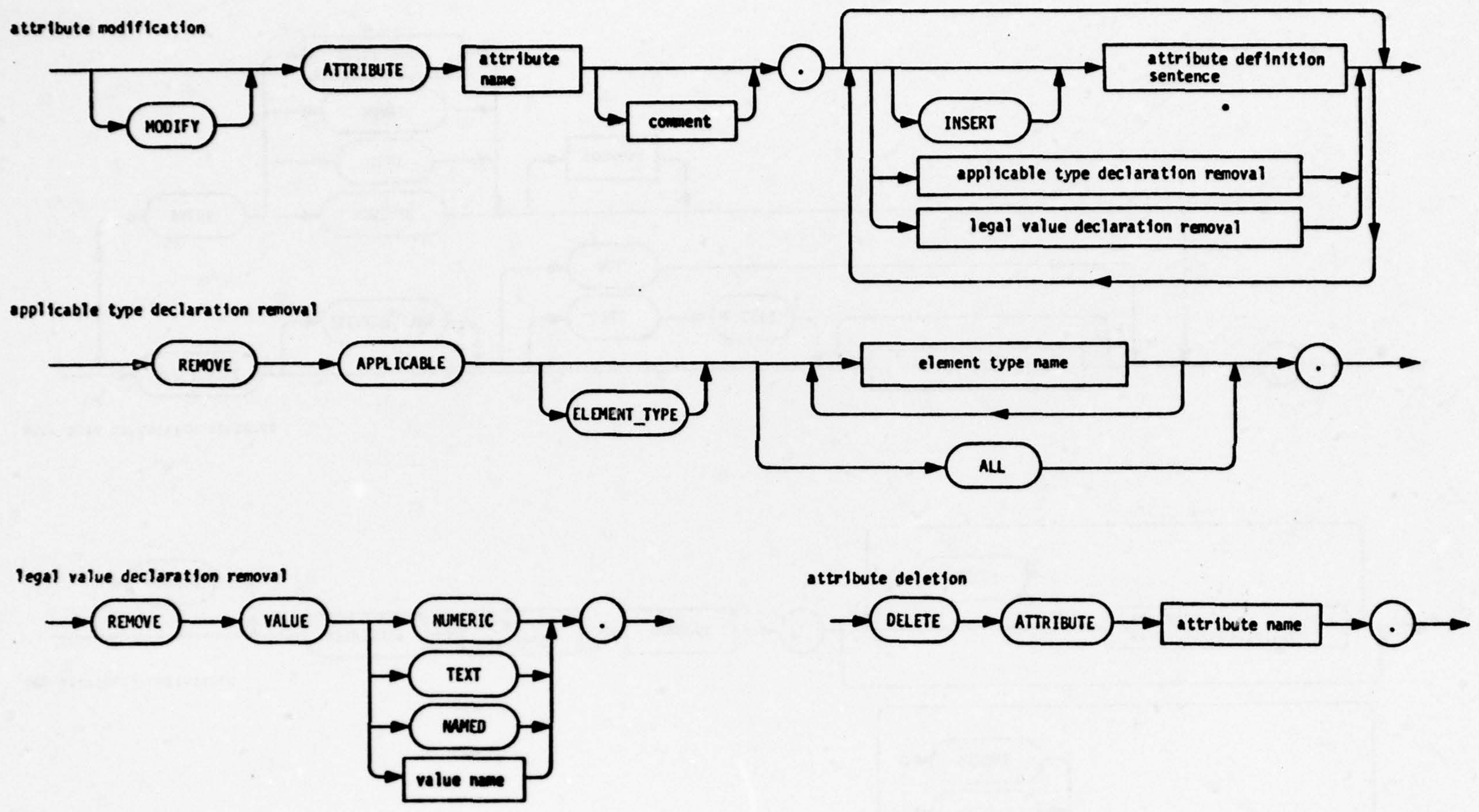


Figure J-1 RSL Extension Syntax Diagrams (Continued)

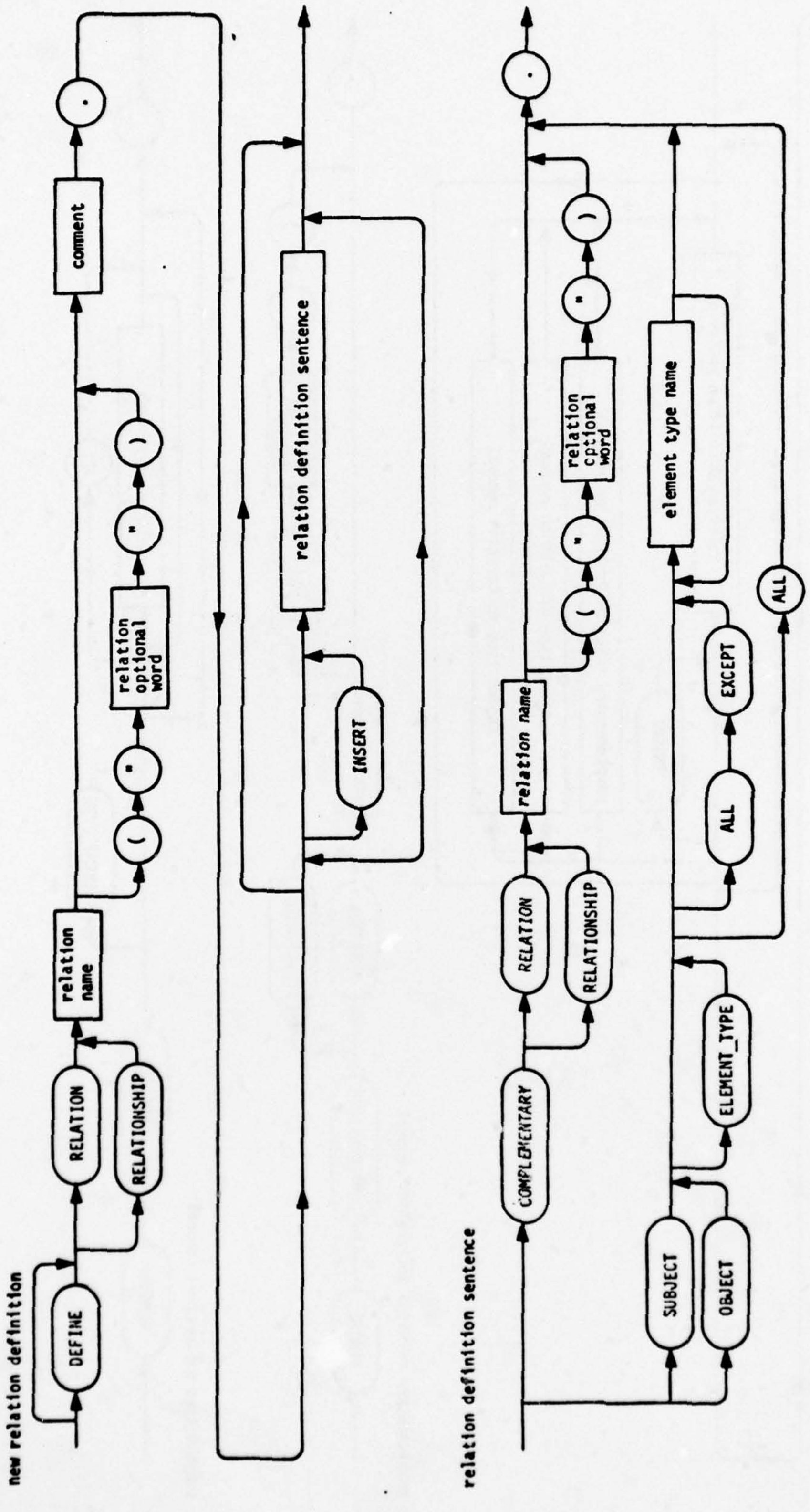
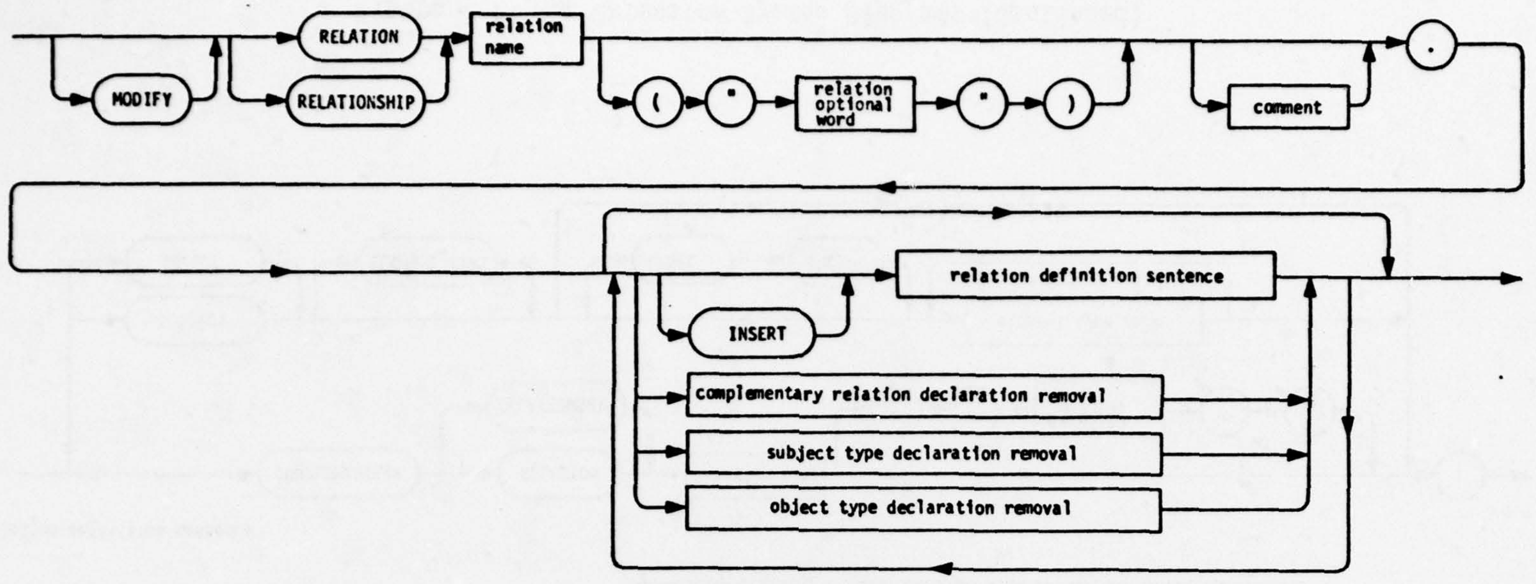
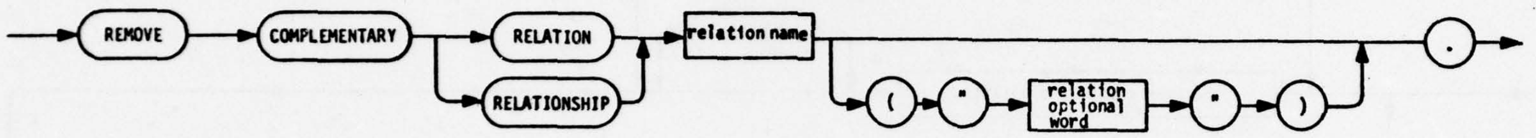


Figure J-1 RSL Extension Syntax Diagrams (Continued)

relation modification


MODIFY
RELATION
RELATIONSHIP
relation name
(
"
relation optional word
"
)
comment
.
relation definition sentence
INSERT
complementary relation declaration removal
subject type declaration removal
object type declaration removal

complementary relation declaration removal



subject type declaration removal

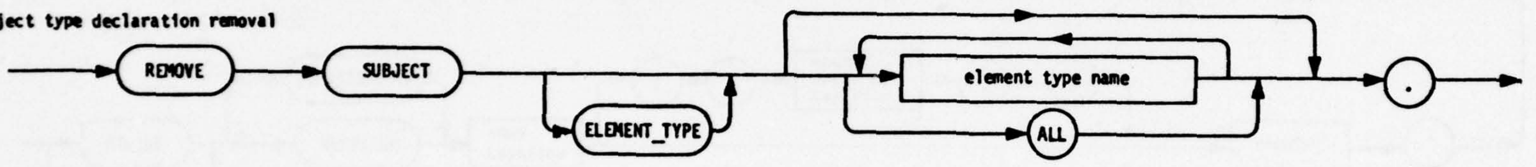
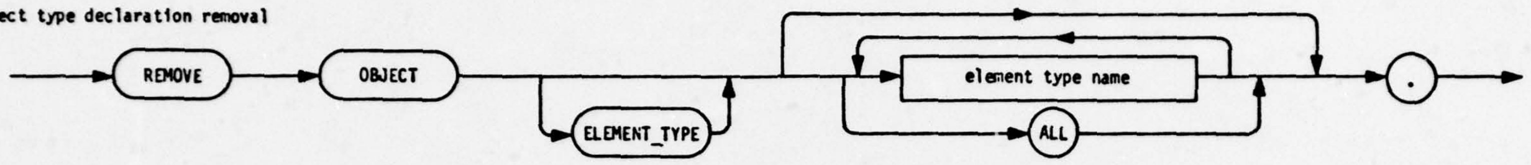


Figure J-1 RSL Extension Syntax Diagrams (Continued)

object type declaration removal



relation deletion

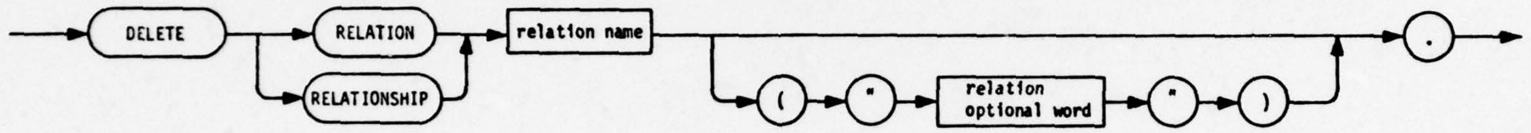


Figure J-1 RSL Extension Syntax Diagrams (Continued)

REFERENCES

1. "Process Design Methodology Design System Specification", Volumes I-V, Texas Instruments, Incorporated, Document No. H750502-1B, September 1976.

2. K. Jensen and N. Wirth, PASCAL: User Manual and Report, Second Edition, Springer-Verlag New York Inc., New York, N. Y., 1975.

3. "ASC Job Specification Language Reference Manual", Texas Instruments, Incorporated, Document No. 930038-4, May, 1976.

4. "Control Data CYBER 70/Model 76 Computer System, 7600 Computer System, SCOPE 2.1 Reference Manual", Control Data Corporation, Publication No. 60342600, (Revised) May 1975.

5. "ASC Linkage Editor User's Guide", Texas Instruments, Incorporated, Document No. 930057-2, April 1976.

6. "Control Data CYBER 170 Series/Models 172, 173, 174, 175, CYBER 70 Series/Models 72, 73, 74, 76, 6000 Series, 7600 Computer Systems, Loader Reference Manual", Control Data Corporation, Publication No. 60344200, (Revised) September 1975.